DTIC FILE COPY

January 1990

UILU-ENG-90-2201
ACT-110

COORDINATED SCIENCE LABORATORY
College of Engineering
Applied Computation Theory

AD-A217 685

COMPUTATIONAL COMPLEXITY OF RANDOM ACCESS MODELS

David Ralph Luginbuhl

DTIC
ELECTE
FEB 06 1990
S D
E

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Approved for Public Release. Distribution Unlimited.

90 02 06 189

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-90-2201 (ACT #110)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION Air Force Institute of Technology
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Ave. Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) Wright-Patterson AFB Ohio 45433
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Air Force Institute of Technology 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER ---
8c. ADDRESS (City, State, and ZIP Code) Wright-Patterson AFB Ohio 45433	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

Computational Complexity of Random Access Models

12. PERSONAL AUTHOR(S)
Luginbuhl, David Ralph

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	January 1990	102

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Computational complexity, random access model, random access machine, pointer machine, storage modification machine, Turing machine, multidimensional Turing machine,

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The relative power of several computational models is considered. These models are the Turing machine and its multidimensional variant, the random access machine (RAM) , the tree machine, and the pointer machine. The basic computational properties of the pointer machine are examined in more detail. For example, time and space hierarchy theorems for pointer machines are presented.

Every Turing machine of time complexity t and space complexity s can be simulated by a pointer machine of time complexity $O(t)$ using $O(s/\log s)$ nodes. This strengthens a similar result by van Emde Boas (1989). Every alternating pointer machine of time complexity t can be simulated by a deterministic pointer machine using $O(t/\log t)$ nodes. Other results concerning nondeterministic and alternating pointer machines are presented.

(continued)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

19. Abstract (continued)

Every tree machine of time complexity t can be simulated on-line by a log-cost RAM of time complexity $O((t \log t)/\log \log t)$. This simulation is shown to be optimal using the notion of incompressibility from Kolmogorov complexity (Solomonoff, 1964; Kolmogorov, 1965).

Every d-dimensional Turing machine of time complexity t can be simulated on-line by a log-cost RAM running in time $O(t(\log t)^{1-(1/d)}(\log \log t)^{1/d})$. There is a log-cost RAM R running in time t such that every d-dimensional Turing machine requires time $\Omega(t^{1+(1/d)}/(\log t(\log \log t)^{1+(1/d)}))$ to simulate R on-line. Every unit-cost RAM of time complexity t can be simulated on-line by a d-dimensional Turing machine in time $O(t(n)^2 \log t(n))$.

18. Kolmogorov complexity, simulation

COMPUTATIONAL COMPLEXITY OF RANDOM ACCESS MODELS

BY

DAVID RALPH LUGINBUHL

B.A., Florida State University, 1979
M.S., Florida State University, 1981

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1990

Urbana, Illinois

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

Abstract

The relative power of several computational models is considered. These models are the Turing machine and its multidimensional variant, the random access machine (RAM) , the tree machine, and the pointer machine. The basic computational properties of the pointer machine are examined in more detail. For example, time and space hierarchy theorems for pointer machines are presented.

Every Turing machine of time complexity t and space complexity s can be simulated by a pointer machine of time complexity $O(t)$ using $O(s/\log s)$ nodes. This strengthens a similar result by van Emde Boas (1989). Every alternating pointer machine of time complexity t can be simulated by a deterministic pointer machine using $O(t/\log t)$ nodes. Other results concerning nondeterministic and alternating pointer machines are presented.

Every tree machine of time complexity t can be simulated on-line by a log-cost RAM of time complexity $O((t \log t)/\log\log t)$. This simulation is shown to be optimal using the notion of incompressibility from Kolmogorov complexity (Solomonoff, 1964; Kolmogorov, 1965).

Every d-dimensional Turing machine of time complexity t can be simulated on-line by a log-cost RAM running in time $O(t(\log t)^{1-(1/d)}(\log\log t)^{1/d})$. There is a log-cost RAM R running in time t such that every d-dimensional Turing machine requires time $\Omega(t^{1+(1/d)}/(\log t(\log\log t)^{1+(1/d)}))$ to simulate R on-line. Every unit-cost RAM of time complexity t can be simulated on-line by a d-dimensional Turing machine in time $O(t(n)^2 \log t(n))$.

Acknowledgements

Many thanks to the United States Air Force and to the Air Force Institute of Technology (AFIT) for their support in terms of finance and time. Such a deal!! Special thanks to Captain Gaylene Ujcik, my program manager at AFIT, for responding immediately whenever I had a question or request. Thanks also to AFIT Public Affairs for screening all my material so that the Air Force would not take the blame for any spurious results or embarrassing statements. It's nice to be loved and trusted.

I am grateful to the administrative staffs of the Coordinated Science Laboratory and Beckman Institute; they have been extremely pleasant and helpful.

I also want to thank my friends for helping me preserve my sanity while I was here, especially to those involved with Graduate InterVarsity Christian Fellowship. There are many, but some are very special to me: Diane Cook, Chip Mayse (UB), Leann and Dan Waldron, Shirley Faughn, Christie and Duane Nieuwsma, Deb Lundberg Windes(LB), and Gary Murakami. I love you all and will miss you very much. Thanks as well to my officemates Marsha and Mike for hours of scholarly discussions at CSL and Beckman (yeah, right!), and thanks to anyone I played volleyball with.

Thanks to my sister Kristen for long-distance listening. You know that I love you.

Of course, my Mom and Dad have provided more moral support than anyone, and their unconditional love has kept me going. I thank God continually for my parents.

My advisor, Professor Michael C. Loui has made this entire experience worthwhile and even fun!! I thank him for his motivation, guidance, and red ink, but most of all for his confidence in me. He is everything I expected in an advisor, and I will strive to do as well for my future students as he did for me.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Since the introduction of the Turing machine, computer scientists have devised many abstract models of computation to study different aspects of computation. They began by adding features to the Turing machine so that describing a particular Turing machine program was not so cumbersome. Later, models with random access storage were introduced as a more realistic alternative to the Turing machine. More exotic models have appeared in the literature as well.

With the introduction of each new computational model there comes the question of its quantitative relationship to its rivals. It is clear that all the most prominent models (excluding models such as the finite state automaton and the pushdown automaton) accept the recursively enumerable languages, but they do so with varying degrees of time and space efficiency.

The purpose of this thesis is to specify relationships between models of computation. In particular, we describe the relative computational power between specific models by designing and analyzing simulations of one model by another model. We show in some cases that there are lower bounds to the speed of simulations of one machine by another.

1.2 Summary of Results

The models we consider are the Turing machine and its multidimensional variant, the random access machine (RAM), the tree machine, and the pointer machine. We examine in more

detail the basic computational properties of the pointer machine, a newer model that has been neglected in the literature but seems to have some interesting properties.

We begin our review of the pointer machine by showing that space compression is possible. We then describe how space equivalence of pointer machines to other models of computation depends on the definition of pointer machine space complexity. We show that every Turing machine of time complexity t and space complexity s can be simulated by a pointer machine of time complexity $O(t)$ using $O(s/\log s)$ nodes. This strengthens a similar result by van Emde Boas (1989).

We give time and space hierarchy theorems for pointer machines. With respect to the time and space hierarchies, pointer machines are similar to Turing machines and RAMs. We show that every alternating pointer machine of time complexity t can be simulated by a deterministic pointer machine using $O(t/\log t)$ nodes. We present other results concerning nondeterministic and alternating pointer machines.

We describe how every tree machine of time complexity t can be simulated on-line by a log-cost RAM of time complexity $O((t \log t)/\log\log t)$. Using the notion of incompressibility from Kolmogorov complexity (Li and Vitanyi, 1988), we show that our simulation method is optimal. This appears to be the first application of Kolmogorov complexity to sequential RAMs. It is significant because few algorithms have been shown to be optimal.

Using similar techniques, we show that every d-dimensional Turing machine of time complexity t can be simulated on-line by a log-cost RAM running in time $O(t(\log t)^{1-(1/d)}(\log\log t)^{1/d})$. For $d = 1$, the running time is $O(t \log\log t)$, which is the same as the result of Katajainen *et al.* (1988).

For simulations of RAMs by multidimensional Turing machines, we show that there is a log-cost RAM R running in time t such that every d-dimensional Turing machine requires time $\Omega(t^{1+(1/d)}/(\log t(\log\log t)^{1+(1/d)}))$ to simulate R on-line. We describe how every unit-cost RAM of time complexity t can be simulated on-line by a d-dimensional Turing machine in time $O(t(n)^2 \log t(n))$. We also give a lower bound for on-line simulations of unit-cost RAMs by multidimensional Turing machines.

1.3 Motivation

Our research is significant for a number of reasons. A general simulation provides automatic transformation of algorithms for one model into algorithms for another model. This is especially important when one of the models is a Turing machine or RAM, since these machines are clearly the standard models of computation: the Turing machine for its historical significance, the RAM because it most closely resembles a real computer.

Another reason to examine relationships between these models of computation is to determine how architectural enhancements (for example, multidimensional tapes on a Turing machine) affect the speed of a model.

A related issue is how alterations in the definition of time and space measures for a model affect the model's complexity. We are particularly interested in the tradeoff between log-cost RAMs with the standard instruction set and unit-cost RAMs with a weaker instruction set (allowing only the successor function). We have tried to determine their relationship directly, but we have also examined their relationship by looking at how they each relate to other models.

An additional motivation for research into simulations between these models concerns the idea of data structure representation. The simulations provide insight into efficient embeddings

between data structures. For instance, an optimal simulation of a multidimensional Turing machine by a RAM may indicate how best to represent an array in a set of registers.

It is clear that researchers will continue to use the models investigated in this thesis. We feel that our results add significantly to the knowledge base of computational modeling.

1.4 Overview

In Chapter 2, we introduce the machines we investigate, providing detailed descriptions of each model, as well as definitions of their time and space complexities. In Chapter 3, we survey research pertinent to our study and review previous results involving these models.

We investigate the pointer machine in Chapter 4. We present some basic computational properties of the pointer machine. Chapter 4 also contains our results on space equivalence of pointer machines, Turing machines, and RAMs, as well as some theorems about nondeterministic and alternating pointer machines.

Chapter 5 features our on-line simulation of a tree machine by a log-cost RAM. We describe the simulation and show that it is optimal. In Chapter 6, we exhibit on-line simulations between multidimensional Turing machines and RAMs, in both directions, and present some lower bounds.

In Chapter 7, we present open problems suggested by the results of Chapters 4, 5, and 6, and we describe the significance of their solutions.

A preliminary version of Chapter 4 appears in a Coordinated Science Laboratory technical report (Luginbuhl and Loui, 1988). Preliminary versions of Chapter 5 and Section 6.1 also appear in a technical report (Loui and Luginbuhl, 1989).

Chapter 2

Definitions and Notation

In this chapter, we introduce the models of computation we consider. We also describe terms and notation we use throughout the thesis. All logarithms in this thesis are taken to base 2. For simplicity, we omit floors and ceilings.

2.1 Turing Machines

We assume the reader is familiar with the Turing model, its basic variations, and the time and space measures of the Turing machine as described by Hopcroft and Ullman (1979). Because many of our results concern multidimensional Turing machines, we describe them here in more detail.

A multihead d-dimensional Turing machine consists of a finite control and a finite number of d-dimensional worktapes, each with at least one worktape head. A d-dimensional worktape comprises an infinite number of cells, each of which is assigned a d-tuple of integers called the *coordinates* of the cell; for instance, the coordinates of cell x are $(x_1, x_2, \ldots, x_d)$. The coordinates of adjacent cells differ in just one component of the d-tuple by ± 1. Call the cell with coordinates $(0, 0, \ldots, 0)$ the *origin*.

At each step of the computation, the machine reads the symbols in the currently accessed input and worktape cells, (possibly) writes symbols on the currently accessed output and work-

tape cells, (possibly) shifts the input head, and shifts each worktape head in one of $2d + 1$ directions – either to one of $2d$ adjacent cells or to the same cell.

A *box* is a set of cells that form a d-dimensional cube. The *volume* of a box is the number of cells it contains. The *base cell* of a box is the cell within the box with the smallest coordinates. The *distance* between two cells is the sum of the absolute values of the differences between their corresponding coordinate components; this is sometimes called the L^1-*distance* or *rectilinear distance.*

Lemma 2.1 *Let M be a d-dimensional Turing machine with $d \geq 2$. Let a worktape of M have a box B of volume v with the origin as B's base cell, and let x be a cell within box B. If the coordinates of x are written on a separate worktape, then M can access cell x in time $O(v^{1/d})$.*

Proof. M moves a worktape head to the origin (the boundaries of B are specially marked so that the origin can be found). M then moves this worktape head in the first direction the distance specified by the corresponding component x_1 of x's coordinates, decrementing that component as the head moves in B. M does the same for each component, until the head arrives at cell x. Decrementing a counter specifying value v' takes time $O(v')$. Since each of x's d components specifies a value of at most $v^{1/d}$, decrementing each component takes time at most $dO(v^{1/d}) = O(v^{1/d})$ (since d is a constant). The distance from x to the origin is at most $dv^{1/d}$, so M moves the head across $O(v^{1/d})$ cells. Thus the total time to move the head to x is $O(v^{1/d})$. □

2.2 Tree Machines

A *tree machine*, a generalization of a Turing machine, has a storage structure that consists of a finite collection of complete infinite rooted binary trees, called *tree worktapes.* Each cell of a

worktape can store a 0 or 1. Each worktape has one head. A worktape head can shift to a cell's parent or to its left or right child. Initially, every worktape head is on the root of its worktape, and all cells contain 0.

Let W be a tree worktape. We fix a natural bijection between the positive integers and cells of W. We refer to the integer corresponding to a particular cell as that cell's *location*. Write cell(b) for the cell at location b. Define cell(1) as the root of W. Then cell($2b$) is the left child of cell(b) and cell($2b+1$) is the right child of cell(b).

Each step of a tree machine consists of reading the contents of the worktape cells and input cell currently scanned, writing back on the same worktape cells and (possibly) to the currently accessed output cell, and (possibly) shifting each worktape head and the input head. When the tree machine writes on the output tape, it also shifts the output head.

The *time complexity* $t(n)$ of a tree machine is defined in the natural way. The *space complexity* $s(n)$ of a tree machine is the total number of distinct cells visited by worktape heads. Clearly, every tree machine of space complexity $s(n)$ can be simulated by a Turing machine of space complexity $s(n)$, and vice-versa.

The *depth complexity* of a tree machine is $d(n)$ if every worktape head remains within distance d of the root of its worktape on every input of size n. It is possible to limit the depth complexity of a tree machine with respect to its time complexity:

Theorem 2.2 (Paul and Reischuk, 1981; Loui, 1984a) *Every tree machine running in time $t(n)$ can be simulated on-line by a tree machine running in time $O(t(n))$ and depth $O(\log t(n))$.*

2.3 Random Access Machines (RAMs)

The random access machine (RAM) (Aho *et al.*, 1974; Cook and Reckhow, 1973; Katajainen *et al.*, 1988) consists of the following: a finite sequence of labeled instructions; a memory consisting of an infinite sequence of registers, indexed by nonnegative integer addresses (register $r(j)$ has address j); and a special register AC, called the *accumulator*, used for operating on data. Each register, including AC, holds a nonnegative integer; initially all registers contain 0. Let $\langle x \rangle$ denote the contents of register $r(x)$ and $\langle AC \rangle$ denote the contents of AC. Each cell on the input and output tapes contains a symbol from a finite input/ouput alphabet. The following RAM instructions are allowed:

input. Read the current input symbol into AC and move the input head one cell to the right.

output. Write the binary representation of $\langle AC \rangle$ onto the output tape.

jump θ. Unconditional transfer of control to instruction labeled θ.

jgtz θ. Transfer control to instruction labeled θ if $\langle AC \rangle > 0$.

load $=C$. Load integer C into AC.

load j. Load $\langle j \rangle$ into AC.

load $*j$. (Load indirect) Load $\langle \langle j \rangle \rangle$ into AC.

store j. Store $\langle AC \rangle$ into $r(j)$.

store $*j$. (Store indirect) Store $\langle AC \rangle$ into register $r(\langle j \rangle)$.

add j. Add $\langle j \rangle$ to $\langle AC \rangle$ and place result in AC.

sub j. If $\langle j \rangle > \langle AC \rangle$, then load 0 into AC; otherwise, subtract $\langle j \rangle$ from $\langle AC \rangle$ and place result in AC.

The *length* of a nonnegative integer i is the minimum positive integer w such that $i \leq 2^w - 1$ (approximately the logarithm of i).

We consider two time complexity measures for RAMs, based on the cost of each RAM instruction. For the *unit-cost RAM*, we charge each instruction one unit of time. For the *log-cost RAM*, we charge each instruction according to the *logarithmic cost criterion* (Cook and Reckhow, 1973): the time for each instruction is the sum of the lengths of the integers (addresses and register contents) involved in its execution. The *time complexity* $t(n)$ of a RAM is the maximum total time used in computations on inputs of length n. It is possible, of course, to define time complexity in other ways; e.g., we could charge some other function $f(j)$ for access to register j (Aggarwal *et al.*, 1987).

In our simulations involving RAMs, we group the registers into a finite number of memories, each memory containing an infinite number of registers. This does not increase the cost in time by more than a constant factor, since we could simply interleave these memories into one memory (Katajainen *et al.*, 1988).

We discuss the space complexity of RAMs in Section 3.5.

Two RAM operations used often in this thesis are the *pack* and *unpack* operations. Let $r_1, r_2, \ldots, r_b$ be contiguous registers in RAM R's memory containing, respectively, $x_1, x_2, \ldots, x_b$, where each x_i is a single bit. R *packs* $r_1, r_2, \ldots, r_b$ by computing the single b-bit value $2^{b-1}x_1 + 2^{b-2}x_2 + \ldots + x_b$ and placing this value into the accumulator (see Figure 2.1 for an example). The

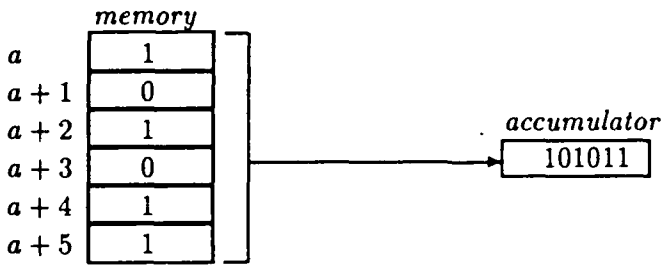


Figure 2.1: Packing six bits into the accumulator

unpack operation is the inverse of the pack operation; R takes a single value in the accumulator and stores its bits into contiguous registers. Each operation has as parameters the beginning and ending addresses of the registers involved in the operation.

We use a technique of Katajainen *et al.* (1988) to pack and unpack registers. This divide-and-conquer strategy involves precomputed shift tables:

Lemma 2.3 (Katajainen *et al.*, 1988) *If the proper tables are available, then it is possible to pack u bits into the accumulator, and to unpack a u-bit string into memory, both in $O(u \log u)$ time on a log-cost RAM.*

Lemma 2.4 (Katajainen *et al.*, 1988) *The tables necessary for Lemma 2.3 can be built in $O(u2^u)$ time on a log-cost RAM.*

2.4 Unit-cost Successor RAMs

A variation of the RAM that we consider is the *successor* RAM (SRAM) (see Schönhage, 1980). The SRAM is defined almost exactly like the RAM, except that instead of add and

subtract instructions, the SRAM has the *successor* instruction, which adds 1 to the value in the accumulator, and the *empty* instruction, which loads 0 into the accumulator. We are particularly interested in the SRAM with the unit-cost time measure, because of the relationship between the unit-cost SRAM and the pointer machine (see Section 3.4).

2.5 Pointer Machines

The following definition for pointer machines is drawn from the pointer machine definitions of Schönhage (1980) and Halpern *et al.* (1986).

A Δ-*structure*, which provides the storage for the pointer machine, is a directed graph consisting of *nodes* (vertices) and *pointers* (edges). Each node has a finite number of outgoing pointers, and each pointer from a node has a distinct label. The labels are symbols from the finite pointer alphabet Δ. At any time, one node, designated the *center*, is used to access the Δ-structure. We refer to the center node as x_0.

We describe an instantaneous configuration of the Δ-structure by a set of pointer mappings: for all $\delta \in \Delta$, there is a pointer mapping $p_\delta : X \to X$, where X is the set of nodes; $p_\delta(x) = y$ means the δ pointer from node x points to node y. From these pointer mappings, we can recursively define the mapping $p^* : \Delta^* \to X$:

$$p^*(\lambda) = x_0, \qquad \text{(where } \lambda \text{ is the empty word)}$$

$$p^*(W\delta) = p_\delta(p^*(W)), \qquad \text{for all } \delta \in \Delta, W \in \Delta^*.$$

The pointer machine also has a separate read-only input tape containing symbols from an input alphabet Σ. For simplicity, we consider only pointer machines that accept or reject their input. With the addition of a write-only output tape, we could also consider pointer machines that produce output.

A pointer machine has a finite sequence of program instructions, each with some distinct instruction label. We define the following pointer machine instructions:

accept. Self-explanatory. Computation halts.

reject. Self-explanatory. Computation halts.

create W, where $W \in \Delta^*$. Create a new node x' in the Δ-structure. If $W = U\delta$, where $\delta \in \Delta$ and $U \in \Delta^*$, then set the δ pointer of node $p^*(U)$ to point to x'. For all $\gamma \in \Delta$, $p_\gamma(x')$ is set to x_0.

center W, where $W \in \Delta^*$. Make the node $p^*(W)$ the new center.

assign $W := V$, where $W,V \in \Delta^*$. If $W = U\delta$, then set the δ pointer of $p^*(U)$ to point to $p^*(V)$.

if $W = V$ **go to** μ, where $W,V \in \Delta^*$ and μ is an instruction label. If $p^*(W) = p^*(V)$, then pass control to the instruction labeled μ. Otherwise, execute the next instruction.

if *input* $= \sigma$ **go to** μ, where μ is an instruction label. If the input symbol is σ, then pass control to the instruction labeled μ. Otherwise, execute the next instruction.

move ρ, where $\rho \in$ {left, right}. Move the input tape head one square in the direction indicated by ρ.

The pointer machine starts with the input head on the leftmost nonblank input symbol and one node in the Δ-structure. We call this node, which is the center when computation begins, the *initial node.*

The *time* consumed by the pointer machine is the number of instructions executed before halting. We consider both unit-cost and logarithmic-cost space measures. Define *mass* to be the number of **create** instructions executed before halting, i.e., the number of nodes created during execution. Mass was introduced as a measure of space by Halpern *et al.* (1986). Define the *capacity* of a computation to be $dn \log n$, where n is the number of nodes created, and d is the number of pointers per node ($d = |\Delta|$).

The idea for considering capacity as a space measure comes from Borodin *et al.* (1981). With n nodes there are at most n^{dn} possible configurations of the Δ-structure. Borodin *et al.* (1981) defined control space (capacity) as $\log(Q)$, where Q is the number of possible configurations, so our definition of pointer machine capacity is reasonable. Since the standard definition of Turing machine space does not account for the size of the tape alphabet, we could also define a logarithmic space measure for pointer machines without considering the size of the pointer alphabet. For this thesis, however, we use the capacity measure as defined.

Proposition 2.5 *Every pointer machine of mass complexity $s(n)$ has capacity complexity $O(s(n)\log s(n))$ and every pointer machine of capacity complexity $s(n)$ has mass complexity $O(s(n)/\log s(n))$.*

2.6 Nondeterministic and Alternating Machines

Although most of our results involve sequential models of computation, there are a few theorems about nondeterministic and alternating machines. We assume the reader is familiar with nondeterministic Turing machines as described by Hopcroft and Ullman (1979).

The concept of *alternation* (Chandra *et al.*, 1981) builds on the idea of nondeterminism. In an alternating Turing machine, as in a nondeterministic Turing machine, each configuration

can reach several successor configurations in one step. Each state is *universal* or *existential.* A configuration of the alternating machine is universal if the state is universal, and it is existential if the state is existential. An existential configuration is accepting if at least one of its successor configurations is accepting. A universal configuration is accepting only if all of its successor configurations are accepting. A configuration with no successor configuration is accepting if the state is accepting, and vice-versa. Thus a nondeterministic Turing machine is an alternating Turing machine with all existential configurations.

Nondeterministic and alternating pointer machines are defined analogously.

2.7 Miscellaneous Definitions

Let $resource_1$ be time or space and $resource_2$ be time or space. We say that machine M of $resource_1$ complexity r is simulated by machine M' in $resource_2$ $f(r)$ if for every input string, M' produces the same output as M in $resource_2$ $f(r)$. We call M' the *host* and M the *guest* in the simulation.

We say M is simulated by a machine M' *on-line* in time $f(t)$ if for every time step t_i where M reads/writes an input/output symbol, there is a corresponding time step t_i' where M' reads/writes the same symbol, and $t_i' \leq f(t_i)$. If M' simulates M in time $f(t)$, but the simulation is not on-line, then we say that M' simulates M *off-line.* Clearly, if M' simulates M on-line in time $f(t)$, then we can modify M' to simulate M off-line in time $f(t)$. The converse may not be true. The distinction between on-line and off-line simulations is meaningless if the simulated machine reads all of its input before writing any output (e.g., a machine that only accepts or rejects its input).

We say M is simulated by M' in *real time* if there is a constant c such that the following holds: if M reads/writes an input/output symbol at time steps $t_0 < t_1 < \cdots < t_\ell$, then M' reads/writes the same symbol at time steps $t'_0 < t'_1 < \cdots < t'_\ell$, and for $1 \leq i \leq \ell$, $t'_i - t'_{i-1} \leq c(t_i - t_{i-1})$. If machine M simulates M' in real-time and M' simulates M in real-time, then M and M' are *real-time equivalent.*

Chapter 3

Literature Review

This chapter surveys research into the relative power of computational models. We begin with a review of results on Turing machines and RAMs.

3.1 Turing Machines

Early results in computational complexity involved the deterministic Turing machine. Hopcroft and Ullman (1979) presented many of these results, such as linear speedup, space compression, and time and space hierarchies.

Hennie and Stearns (1966) showed that Turing machines running in time t accept more languages than Turing machines running in time $o(t/\log t)$. This was an improvement of an earlier time hierarchy presented by Hartmanis and Stearns (1965b). Cook and Reckhow (1973) presented a tighter hierarchy for log-cost RAMs; specifically, RAMs running in time t accept more languages than RAMs running in time $o(t)$. Paul (1979) substantially tightened the time hierarchy for k-tape Turing machines, for fixed $k \geq 2$; he showed that the class of languages accepted by k-tape machines in time t is strictly contained in the class of languages accepted by k-tape machines in time $o(t \log^* t)$. Later, Fürer (1984) showed that the time hierarchy is as tight for multitape Turing machines (with a fixed number of tapes) as it is for RAMs.

The definition of the Turing machine model has been modified in many ways to investigate how different enhancements of the model affect its computational power. One such variation

already mentioned is the multitape Turing machine. Hennie and Stearns (1966) showed that a k-tape machine can be simulated by a 2-tape machine within a logarithmic time factor. Paul *et al.* (1981) proved that a k-tape machine cannot be simulated in real time by a Turing machine with fewer than k tapes. Their proof relied on information-theoretic techniques from Kolmogorov complexity, which has since been used to prove lower bound results for other on-line simulations (e.g., Loui, 1983), including some results in this thesis. We discuss Kolmogorov complexity in more detail in Subsection 5.2.2.

Another variant of the Turing machine is the *multidimensional Turing machine*. Hennie (1966) and Grigor'ev (1977) showed that for all $e > d$, a d-dimensional Turing machine requires time $\Omega(t^{1+(1/d)-(1/e)})$ time to simulate on-line an e-dimensional Turing machine running in time t. Pippenger (1982) showed that this lower bound holds even if the simulating machine is probabilistic (a probabilistic machine's computation path is determined by a series of "coin flips"). Pippenger and Fischer (1979) achieved this lower bound for $d = 1$: they showed that every e-dimensional Turing machine running in time t can be simulated by a one-dimensional Turing machine in time $O(t^{2-(1/e)})$. Loui (1982) presented a near-optimal upper bound for $d \geq 2$ by showing that every e-dimensional Turing machine of time complexity t can be simulated on-line by a deterministic d-dimensional Turing machine in time $O(t^{1+(1/d)-(1/e)}(\log t)^{O(1)})$. Pippenger (1982) presented simulations of e-dimensional Turing machines of time complexity t by *probabilistic* d-dimensional Turing machines in time $O(t^{1+(1/d)-(1/e)}(\log t)^{1/d})$.

3.2 Random Access Machines (RAMs)

Research into the complexity of RAMs has included investigation of how different definitions of the time measure and different instruction sets affect the time complexity of the RAM.

Hartmanis (1971) described the Random Access Stored-Program Machine (RASP), a RAM whose program is stored in memory. Storing the program in memory allows for alterations in the program during execution. Cook and Reckhow (1973) compared the time complexity of the RAM with the time complexity of the RASP. They showed that the RAM and the RASP are real-time equivalent.

It is clear that every log-cost RAM can be simulated by a unit-cost RAM in real time. Paul and Reischuk (1981) showed that every log-cost RAM running in time t can be simulated off-line by a unit-cost RAM running in time $O(t/\log\log t)$.

Fischer (1975) showed that unit-cost SRAMs can simulate log-cost RAMs with addition and subtraction in real time. Dymond (1977) proved that every unit-cost SRAM running in time t can be simulated on-line by a log-cost RAM in time $O(t \log t)$. A faster on-line simulation is not known, but Schönhage (1988) exhibited a separation between the log-cost RAM and unit-cost SRAM by showing that a log-cost RAM requires $\Omega(n \log^* n)$ time just to read and store (on-line) an arbitrary input of n bits.

With two such different models of computation (the Turing machine and the RAM), it is natural to examine how they differ in their use of time and space. Cook and Reckhow (1973) investigated relationships between Turing machines and log-cost RAMs. They showed that every Turing machine running in time t can be simulated by a log-cost RAM running in time $O(t \log t)$. They also showed that every log-cost RAM running in time t can be simulated in time $O(t^2)$ by a Turing machine. For unit-cost RAMs running in time t, they gave a simulation by a Turing machine in time $O(t^3)$. Katajainen *et al.* (1988) improved the first result: they showed that every Turing machine of time complexity t and space complexity s can be simulated by a log-cost RAM running in time $O(t \log\log s)$ (hence in time $O(t \log\log t)$). Wiedermann (1983)

improved the simulation of a log-cost RAM by a Turing machine to run in time $O(t^2/\log t)$. Loui (1983) proved that every log-cost RAM running in time t can be simulated by a d-dimensional Turing machine running in time $O(t^{1+(1/d)}/\log t)$.

Hopcroft *et al.* (1975) presented an off-line simulation of Turing machines running in time t by unit-cost RAMs running in time $O(t/\log t)$. The RAM precomputes a table of configurations of the Turing machine and simulates the Turing machine using simple table look-ups. This technique is also used in the famous four Russians algorithm for matrix multiplication (Arlazarov *et al.*, 1970). Galil (1976) extended the result of Hopcroft *et al.* (1975) by showing how the simulation could be converted into an on-line simulation.

3.3 Tree Machines

We have seen how additional tapes or more dimensions enhance the complexity of the Turing machine. Researchers have also investigated the effect of more radical variations of the storage structure on the computational complexity of the Turing machine. One variation is the tree machine, whose storage tape is a complete infinite rooted binary tree.

Clearly, every one-dimensional Turing machine can be simulated by a tree machine in real time. Reischuk (1982) showed that every e-dimensional Turing machine of time complexity t can be simulated on-line by a tree machine in time $O(t5^{e\log^* t})$. It is not known whether this simulation is optimal. Pippenger (1982) showed that every multidimensional Turing machine can be simulated on-line by a *probabilistic* tree machine in linear time.

Pippenger and Fischer (1979) showed that every tree machine of time complexity t can be simulated on-line by a one-dimensional Turing machine in time $O(t^2/\log t)$. Extending this result, Loui (1983) showed that every tree machine running in time t can be simulated by a

d-dimensional Turing machine in time $O(t^{1+(1/d)}/\log t)$. He also used Kolmogorov complexity to show that this simulation is optimal.

Some relationships between tree machines and RAMs have also been established. Paul and Reischuk (1981) showed that every log-cost RAM can be simulated by a tree machine in real-time. They also showed that every tree machine running in time t can be simulated off-line by a unit-cost RAM in time $O(t/\log\log t)$.

3.4 Pointer Machines

Schönhage (1980) introduced the Storage Modification Machine, also known as the pointer machine. He showed that the pointer machine is real-time equivalent to the unit-cost SRAM; therefore, results already mentioned concerning the time complexity of unit-cost SRAMs apply as well to pointer machines, and vice-versa. Besides establishing the relationship between pointer machines and SRAMs, Schönhage also showed that pointer machines can simulate multidimensional Turing machines in real time.

Tables 3.1 and 3.2 contain a summary of the relationships between the various models. These tables also include results from this thesis (with accompanying theorem numbers). In these tables, $X \xrightarrow{f} Y$ means that machine X can be simulated by machine Y in time $O(f)$.

3.5 Space Measures

We have already mentioned the existence of a space hierarchy for Turing machines. There are, of course, other results concerning the use of space on the various computational models. For instance, Slot and van Emde Boas (1988) studied the relationship of Turing machine space to

Host

Guest	**Turing machine**	**d-Turing machine**	**tree machine**
Turing machine		real-time (straightforward)	real-time (straightforward)
e-Turing machine	$\Theta(t^{2-(1/e)})$ (Pippenger & Fischer, 79)	$O(t^{1+(1/d)-(1/e)}(\log t)^{O(1)})$ (Loui, 82) $\Omega(t^{1+(1/d)-(1/e)})$ (Hennie, 66) and (Grigor'ev, 77)	$O(t5^{e \log^* t})$ (Reischuk, 82)
tree machine	$\Theta(t^2/\log t)$ (Pippenger & Fischer, 79)	$\Theta(t^{1+(1/d)}/\log t)$ (Loui, 83)	
log-cost RAM	$O(t^2/\log t)$ (Wiedermann, 83) $\Omega(\frac{t^2}{(\log t(\log\log t)^2)})$ (Theorem 6.4)	$O(t^{1+(1/d)}/\log t)$ (Loui, 83) $\Omega(\frac{t^{1+(1/d)}}{(\log t(\log\log t)^{1+(1/d)})})$ (Theorem 6.4)	real-time (Paul & Reischuk, 81)
unit-cost RAM	$O(t^3)$ (Cook & Reckhow, 73)	$O(t^2 \log t)$ (Theorem 6.5) $\Omega(t^{1+(1/d)}/\log t)$ (Theorem 6.6)	$O(t^2)$ (log RAM $\xrightarrow{t}$ tree mach; unit RAM $\xrightarrow{t^2}$ log RAM)
unit-cost SRAM (pointer machine)	$O(t^2 \log t)$ (Wagner & Wechsung, 86)	$O(t^{1+(1/d)}(\log t)^{1/d})$ (Theorem 6.7) $\Omega(t^{1+(1/d)}/\log t)$ (Theorem 6.8)	$O(t \log t)$ (log RAM $\xrightarrow{t}$ tree mach; unit SRAM $\xrightarrow{t \log t}$ log RAM)

Table 3.1: Simulation time bounds (Turing and tree machines)

Host

Guest	**log-cost RAM**	**unit-cost RAM**	**unit-cost SRAM (pointer machine)**
Turing machine	$O(t \log\log t)$ (Katajainen *et al.*, 88)	$O(t/\log t)$ *off-line:* (Hopcroft *et al.*, 75) *on-line:* (Galil, 76)	real-time (straightforward)
e-Turing machine	$O(\frac{t \log t (\log\log t)^{1/e}}{(\log t)^{1/e}})$ (Theorem 6.1)	$O(t/(\log t)^{1/e})$ *off-line:* (Grigor'ev, 79) *on-line:* (Theorem 6.2)	real-time (Schönhage, 80)
tree machine	$\Theta((t \log t)/\log\log t)$ (Theorem 5.2)	*off-line:* $O(t/\log\log t)$ (Paul & Reischuk, 81) *on-line:* real-time (Theorem 5.1)	real-time (straightforward)
log-cost RAM		*off-line:* $O(t/\log\log t)$ (Paul & Reischuk, 81) *on-line:* real-time (straightforward)	$O(t)$ (Fischer, 75)
unit-cost RAM	$O(t^2)$ (straightforward)		$O(t^2)$ (log RAM $\xrightarrow{t}$ unit SRAM; unit RAM $\xrightarrow{t^2}$ log RAM)
unit-cost SRAM (pointer machine)	$O(t \log t)$ (straightforward); $\Omega((t \log t)/\log\log t)$ (Theorem 5.9)	real-time (straightforward)	

Table 3.2: Simulation time bounds (RAMs)

RAM space. One of their goals was to establish whether a Turing machine using space s could simulate a RAM using space $O(s)$, and vice-versa.

Slot and van Emde Boas considered two logarithmic measures for RAM space: $size_s$, which charges only the length of the contents of each register used, and $size_b$, which accounts for both the contents and the address of each register. They showed that both measures allow for space equivalence of RAMs and Turing machines; however, the proof of space equivalence with $size_s$ relied on a simulation of a RAM by a Turing machine in exponential time. On the other hand, simulations based on $size_b$ used linear space and polynomial time in both directions. Thus the second space measure, which seems more intuitive, also seems to allow for a closer relationship between the two models.

One important issue in computational complexity is the relationship of time and space complexity of a particular model. Hopcroft *et al.* (1977) showed that "space is strictly more powerful as a resource for deterministic multitape Turing machines": they proved that every deterministic multitape Turing machine running in time t can be simulated by a Turing machine using space $t/\log t$. Consequently, by the space hierarchy for Turing machines, the class of languages accepted by a Turing machine in time t is strictly contained in the class of languages accepted in space t. Adleman and Loui (1981) presented an alternative proof of the result of Hopcroft *et al.*

Paul and Reischuk (1981) showed that every d-dimensional Turing machine running in time t can be simulated by another d-dimensional Turing machine in space $t5^{d\log^* t}/\log t$. Pippenger (1982) improved the time-space result for multidimensional Turing machines, showing that there is a simulation that runs in space $t/\log t$.

Model	*Time-Space Relationship*	
Turing machine	$t \rightarrow t/\log t$	(Hopcroft *et al.*, 75)
d-Turing machine	$t \rightarrow t/\log t$	(Pippenger, 82)
log-cost RAM	$t \rightarrow t/\log t$	(Paul and Reischuk, 81)
pointer machine (mass)	$t \rightarrow t/\log t$	(Halpern *et al.*, 86)
pointer machine (capacity)	$t \rightarrow t$	(Proposition 2.5)
tree machine	$t \rightarrow t/\log t$	(Paul and Reischuk, 81)

Table 3.3: Time vs. space on computational models

Paul and Reischuk (1981) also showed that every log-cost RAM running in time t can be simulated by a Turing machine in space $O(t/\log t)$. By space equivalence of RAMs and Turing machines (Slot and van Emde Boas, 1988), we have the same time-space result for RAMs as for Turing machines. Another result of Paul and Reischuk is that every tree machine of time complexity t can be simulated by a Turing machine in space $O(t/\log t)$. By space equivalence of tree machines and Turing machines, we again duplicate the Turing machine time-space result, this time for tree machines.

Halpern *et al.* (1986) investigated the relationship of time and space complexity in pointer machines. They showed that every pointer machine running in time t can be simulated in mass $O(t/\log t)$. Table 3.3 summarizes these time vs. space results ($f \rightarrow g$ means that a machine running in time f can be simulated by a machine running in space $O(g)$).

Van Emde Boas (1989) showed that Turing machines running in space s accept exactly the same languages as a pointer machines using $O(s/\log s)$ nodes; however, his simulation of a Turing machine by a pointer machine uses quadratic time. This thesis contains an improvement to this result – our simulation runs in linear time with the same space complexity as the simulation of van Emde Boas.

3.6 Nondeterminism and Alternation

Paul *et al.* (1983) proved that nondeterministic two-tape Turing machines running in linear time accept more languages than k-tape deterministic Turing machines running in linear time for any constant k. The best known simulation of a nondeterministic Turing machine of time complexity t by a deterministic Turing machine takes time $O(1)^t$ (Hopcroft and Ullman, 1979; Yap, 1987). Unfortunately, none of this work has yet led to an answer to the notorious P $\stackrel{?}{=}$ NP question.

Savitch (1970) used a recursive search of machine configurations to prove that every nondeterministic Turing machine with space complexity s can be simulated by a deterministic Turing machine using space s^2. By Savitch's result we know that the class of languages accepted in polynomial space on a deterministic Turing machine is equivalent to the class of languages accepted in polynomial space on a nondeterministic Turing machine (PSPACE = NPSPACE).

Chandra *et al.* (1981) established the fundamental properties of alternating Turing machines. They showed, for instance, that every nondeterministic Turing machine of space complexity s can be simulated by an alternating Turing machine running in time $O(s^2)$. Conversely, every alternating Turing machine of time complexity t can be simulated by a deterministic Turing machine using space t. An important corollary to these two results is that the class of languages accepted in deterministic polynomial space (PSPACE) is equivalent to the class of languages accepted in alternating polynomial time (APTIME).

For alternating Turing machines, Paul *et al.* (1980a) showed that decreasing the number of tapes does not increase the time complexity; i.e., every alternating Turing machine running in time t can be simulated by a one-tape alternating Turing machine with no time loss. Furthermore, they showed that every nondeterministic Turing machine of time complexity t

can be simulated by an alternating Turing machine running in time $t^{1/2}$. Paul and Reischuk (1980b) proved that every deterministic Turing machine of time complexity t can be simulated by an alternating Turing machine running in time $(t \log\log t)/\log t$. Dymond and Tompa (1985) improved this last result with a simulating alternating Turing machine running in time $t/\log t$.

Chapter 4

Pointer Machines

In this chapter, we present several results concerning both time and space complexity of pointer machines. These results indicate that, in many ways, pointer machines are similar to Turing machines and RAMs.

4.1 Notation

We present a standard notation used in this chapter to describe the complexity classes defined by pointer machines and Turing machines.

Let $\mathcal{M}$ be TM for Turing machines and PM for pointer machines. $\mathcal{M}$-TIME(f) is the class of languages accepted by the machine type specified by $\mathcal{M}$ in time f, where f is a function of the input length. TM-SPACE(f) is the class of all languages accepted by Turing machines in space f. PM-CAPACITY(f) and PM-MASS(f) are the space complexity classes for pointer machines. To specify complexity classes for nondeterministic versions of these machines, we prefix the machine type with N; for alternating machines, A.

4.2 Space Compression

Our first result is a space compression theorem for pointer machines.

Theorem 4.1 *For every constant $c > 0$, every pointer machine with mass complexity $s(n)$ can be simulated by some pointer machine with mass complexity $cs(n)$.*

Proof. We show the case where $c = 1/2$. Consider pointer machine A having mass complexity $s(n)$. We design a pointer machine B that simulates A and has mass complexity $s(n)/2$.

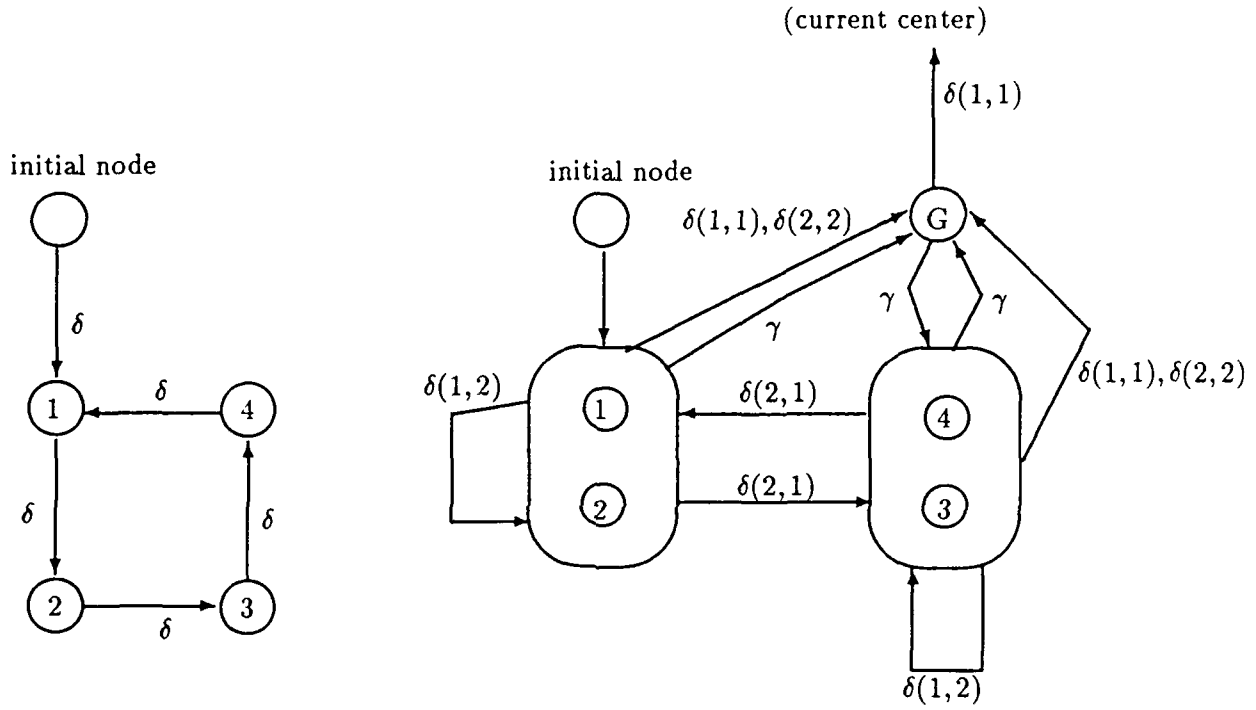
For B to compute with half the number of nodes of A, we encode two nodes of A into one node of B with the addition of several pointers. For every δ in the pointer alphabet of A, the pointer alphabet of B has $\delta(1,1)$, $\delta(1,2)$, $\delta(2,1)$, $\delta(2,2)$. Each node in B corresponds to a pair of nodes in A. The ordered pairs in the pointer notation indicate the original source and destination nodes.

We also create one node (called G) to hold "useless" pointers. And we need a pointer γ that points to G from any node, so G is always accessible. The γ pointer from G always points to the last node created.

We establish the correspondence between nodes of A and nodes of B as follows. Call a node in B a *node-pair* to distinguish it from the pair of nodes in A to which it corresponds. Designate the older node in a pair of nodes in A as node 1 and the other as node 2. If the δ pointer of node 1 in a pair corresponding to node-pair a in B points to node 1 in a pair corresponding to node-pair b, then $p_{\delta(1,1)}(a) = b$ and $p_{\delta(1,2)}(a) = G$. Other cases are handled similarly (see Figure 4.1).

Since we are working with node-pairs in B , we need to designate where the center is within a pair. The structure of B tells us whether a node-pair contains the center of the structure of A by identifying that node-pair as the center. Then we can use a pointer ϕ and two additional nodes H_1 and H_2 in B to tell us whether the center is node 1 or node 2 by having ϕ point to H_1 or H_2, as appropriate.

We initialize B by creating nodes G, H_1, and H_2. After we set their pointers appropriately, we are ready to simulate A.



(nodes numbered in sequence they were created)

Figure 4.1: Reducing the number of nodes by 1/2

Rather than describe the simulation of A in tedious detail, we discuss how to simulate one instruction, **create** δ. The other instructions are simulated analogously.

To simulate **create** δ, we first find the last node-pair created by following the γ pointer of G. Call this node-pair a. We then determine whether node-pair a corresponds to a single node in A (if an odd number of nodes have been created in the execution of A at this point) or to an actual pair. If $p_{\delta(2,1)}(a) = G$ and $p_{\delta(2,2)}(a) = G$, then the node to be created in A is the second in the node-pair a. In this case, we assign the appropriate pointers from the current center to a, and we also assign the appropriate pointer (either $\delta(2,1)$ or $\delta(2,2)$) to the current center from a.

In the case where at least one of $\delta(2,1)$ or $\delta(2,2)$ does not point to G, we must create a new node-pair in B. Then we make the appropriate pointer assignments from this new node-pair to the current center and to G.

With the addition of a few extra pointers, we can eliminate H_1, H_2, and G. We simply encode the information these nodes provide with extra pointers from the initial node. For example, we could substitute pointers ϕ_1 and ϕ_2 for ϕ. One of these two would point to the current center node-pair from the initial node to indicate whether the center is node 1 or node 2.

If A creates $s(n)$ nodes, then B creates $\lceil s(n)/2 \rceil$ nodes (if we eliminate H_1, H_2, and G). We can then generalize the procedure (or continue to apply it repetitively) to achieve space complexity $cs(n)$ for any $c < 1$. □

Note that this simulation does not establish space compression for capacity complexity: if the pointer alphabet size of the original machine is d, then the alphabet size of the simulator is $d(1/c)^2$.

Although space compression is possible using mass as the space measure, it is unclear whether pointer machines also enjoy the linear speedup property of Turing machines.

4.3 Space Requirements and the Invariance Thesis

Slot and van Emde Boas (1988) proposed the following *Invariance Thesis*:

> "There exists a standard class of machine models, which includes among others all variants of Turing machines, all variants of RAMs and RASPs with logarithmic time and space measures, and also the RAMs and RASPs in the uniform time and logarithmic space measure, provided only standard arithmetical instructions of additive type are used. Machine models in this class simulate each other with polynomially bounded overhead in time and constant factor overhead in space."

They showed that using the proper space measure for RAMs, the thesis is true for RAMs and Turing machines in its strictest interpretation; i.e., where the time and space bounds on simulations are met simultaneously. The RAM space measure they proposed charges, for each register used, the logarithm of the register address plus the logarithm of the largest value stored in the register during the computation.

An obvious question is whether this thesis applies as well to pointer machines. Schönhage (1980) presented a real-time equivalence between unit-cost SRAMs, which meet the qualifications of the thesis with respect to time, and pointer machines. So the thesis holds for pointer machines with respect to time complexity. As van Emde Boas (1989) has noted, equivalence

with respect to space complexity depends on the definition of the space measure. The following results make this clearer.

Theorem 4.2 *If $s(n)/\log s(n)$ nodes can be created by a pointer machine in time $O(t)$, then every multitape Turing machine of space complexity $s(n)$ and time complexity $t(n)$ can be simulated by a pointer machine of mass complexity $O(s(n)/\log s(n))$ and time complexity $O(t(n))$. This applies whether both machines are deterministic, nondeterministic, or alternating.*

Proof. We demonstrate how the simulation works for deterministic machines and then show how the result extends to nondeterministic and alternating machines.

The following simulation holds for Turing machines with multiple worktapes; however, for simplicity, we explain how to simulate a Turing machine with a single one-way infinite worktape and a read-only input tape. Without loss of generality, the worktape alphabet of the Turing machine is $\{0,1\}$.

We design a one-to-one correspondence between the storage configurations of a Turing machine M using space s and the configurations of a pointer machine M' with $O(s/\log s)$ nodes: we partition the worktape of M into blocks of size $b = \log(s/\log s)$. With this partitioning, there are $s/b = O(s/\log s)$ blocks. M' represents the tape contents with three node structures: the *tree*, the *blockset*, and the *cache* (see Figure 4.2).

The *tree* is a complete binary tree of height b. The *blockset* consists of $O(s/\log s)$ nodes, $\beta_0, \beta_1, \ldots$, each node representing one block. The contents of a particular block are represented by a pointer to a leaf of the tree. Since the tree has $2^b = s/\log s$ leaves, there is a one-to-one correspondence between the leaves of the tree and the contents of a block. This one-to-one correspondence can be observed by noting that each leaf of the tree can be reached from the root by following a unique sequence of left and right branches. By assigning "0" to each left

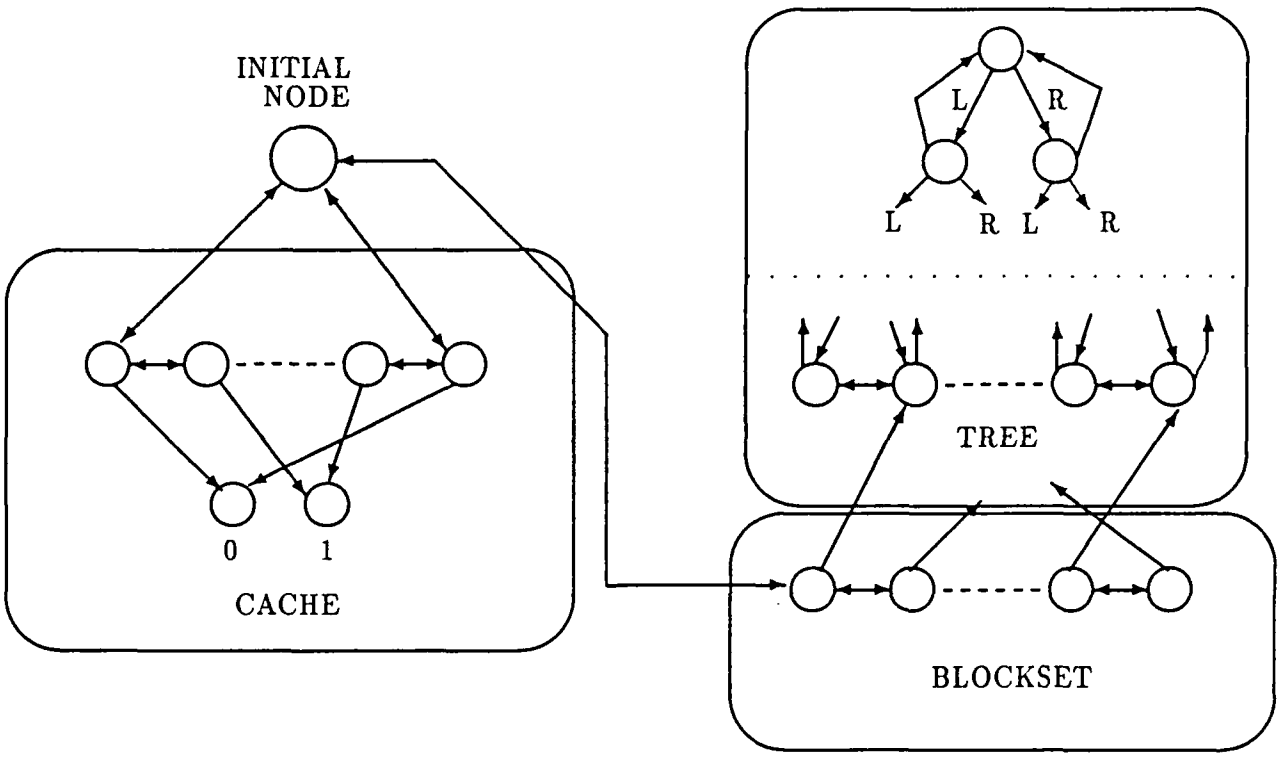


Figure 4.2: Representing a Turing machine with pointer machine nodes

branch and "1" to each right branch, we assign a b-bit number to the leaf. This b-bit number corresponds to a unique configuration of the contents of a block of b cells.

Now consider two adjacent blocks B_i and B_{i+1} of M, such that the worktape head is currently in either B_i or B_{i+1}. B_i and B_{i+1} are represented in the blockset by β_i and β_{i+1}, respectively. During the simulation, M' keeps the contents of β_i and β_{i+1} in the cache.

The *cache* consists of a chain of $2\log(s/\log s)$ nodes and two additional nodes, "0" and "1." Each node in the chain has a pointer to either the "0" or the "1" node, so that the entire chain is a direct representation of two adjacent blocks of M's worktape.

M' decodes a node β_i into the cache as follows: M' finds the tree leaf pointed to by β_i. M' then traces the path of the tree to the root, noting at each tree node whether it was a right or a left child. For each tree node in the path, M' sets a pointer of a node in the cache to the "0" or the "1" node, depending on whether the tree node was a right or left child.

M' encodes the contents of half the cache back to the blockset by following the above steps in reverse.

The simulation proceeds as follows: M' initially builds the blockset, tree, and cache. We assume that blocks are numbered from left to right, that all cells of the worktape of M contain 0, and that M starts with its worktape head on the leftmost tape cell. So M' initially decodes β_0 and β_1 into the cache. M' then begins the actual simulation of M.

Assume β_i and β_{i+1} are decoded in the cache. As long as the tape head remains in B_i and B_{i+1}, M' performs a straightforward simulation of M, using the cache. Finite control and input processing of M are simulated in a straightforward manner using finite control and input processing of M'. When the tape head moves to the right of B_{i+1}, M' encodes B_i back to β_i and decodes β_{i+2} into the cache, shifting to the left the "contents" of B_{i+1} currently in the

cache (by using a few extra pointers to mark the ends and the middle of the cache, M' can "shift" the cache to the left by switching pointers so that the right b nodes are now the left b nodes). If the tape head moves to the left of B_i, then a similar operation occurs. At this time, since the worktape head of M is in the middle of two blocks, M' can simulate at least b steps of M before performing the encoding and decoding operations again.

During the simulation, M' creates $O(s/\log s)$ nodes: $O(s/\log s)$ nodes for the tree and blockset and $O(\log(s/\log s))$ nodes for the cache. Therefore M' simulates M with mass $O(s/\log s)$.

The straightforward simulation in the cache requires a total of $O(t)$ time. The only other time requirement is for encoding and decoding the blocks and shifting. Shifting takes only a constant amount of time. Encoding and decoding can be done in $O(b)$ time, since that is the height of the tree. Because M' maintains two decoded blocks, it performs an encoding and decoding only every $O(b)$ steps, so the total time required for the simulation is $O(t)$.

To prove that this works for nondeterministic and alternating machines, we note that the construction implies an instruction-by-instruction simulation; thus each computation path can be treated as a separate simulation. So each computation path of the Turing machine is simulated in an efficient manner by the corresponding computation path of the pointer machine.

□

Dymond and Cook (1980) use a structure similar to the tree and blockset in their analysis of the relationship between deterministic Turing machines and hardware modification machines (a hardware modification machine is a collection of variably connected, synchronously operating finite state transducers).

Corollary 4.3 *If $s(n)/\log s(n)$ nodes can be created by a pointer machine in time $O(t)$, then every multitape Turing machine of space complexity $s(n)$ and time complexity $t(n)$ can be sim-*

ulated by a pointer machine of capacity complexity $O(s(n))$ and time complexity $O(t(n))$. This applies whether both machines are deterministic, nondeterministic, or alternating.

Proof. By Proposition 2.5, a pointer machine of mass complexity s has capacity complexity $O(s \log s)$. By Theorem 4.2, we know that a Turing machine using space s can be simulated by a pointer machine of mass complexity $s/\log s$. This pointer machine has capacity complexity $O((s/\log s)\log(s/\log s)) = O((s/\log s)\log s) = O(s)$. □

Theorem 4.2 sharpens the following result of van Emde Boas (1989), whose pointer machine simulator used mass $O(s/\log s)$, but time $O(t^2)$ (although his simulation did not require $s/\log s$ to be constructible):

Theorem 4.4 (van Emde Boas, 1989) *Every Turing machine of space complexity $s(n)$ can be simulated by a pointer machine of capacity complexity $O(s(n))$ (hence, of mass complexity $O(s(n)/\log s(n))$) in polynomial time. This applies whether both machines are deterministic or nondeterministic.*

Van Emde Boas also gave a result for simulation in the other direction:

Theorem 4.5 (van Emde Boas, 1989) *Every pointer machine of capacity complexity $s(n)$ (hence, of mass complexity $s(n)/\log s(n)$) can be simulated by a Turing machine in space $s(n)$ in polynomial time. This applies whether both machines are deterministic or nondeterministic.*

Since Slot and van Emde Boas showed that the Invariance Thesis holds in its strictest interpretation (i.e., where time and space bounds are met simultaneously) for Turing machines and RAMs using the $size_b$ space measure, we can establish a relationship between RAMs and pointer machines:

Corollary 4.6 *Every pointer machine can be simulated by a RAM in polynomial time and with constant factor overhead in space (*size_b *space measure for RAMs).*

Corollary 4.7 *Every RAM can be simulated by a pointer machine in polynomial time and with constant factor overhead in capacity (*size_b *space measure for RAMs).*

If we consider capacity as the true measure of space in pointer machines, then we must reevaluate the result of Halpern *et al.* (1986): that every pointer machine of time complexity t can be simulated by a pointer machine of space complexity $O(t/\log t)$. The authors considered mass as the space measure. A different approach is necessary to achieve the same result for capacity, if it is even possible.

Using Theorem 4.4, we obtain a time-space result in the other direction.

Theorem 4.8 *Every pointer machine of capacity complexity $s(n)$ can be simulated by a pointer machine of time complexity $O(nO(1)^{s(n)})$. This applies whether the machines are deterministic or nondeterministic.*

Proof. Let X be D or N.

$$\begin{aligned} \text{XPM-CAPACITY}(s) &= \text{XTM-SPACE}(s) && \text{(Theorem 4.4)} \\ &\subseteq \text{XTM-TIME}(nO(1)^s) && \text{(see Yap, 1987)} \\ &\subseteq \text{XPM-TIME}(nO(1)^s) && \text{(Schönhage, 1980)} \end{aligned}$$

□

Combining Theorem 4.8 and Proposition 2.5, we have:

Corollary 4.9 *Every pointer machine of mass complexity $s(n)$ can be simulated by a pointer machine of time complexity $O(ns(n)^{O(s(n))})$. This applies whether the machines are deterministic or nondeterministic.*

Proof. Let X be D or N.

$$\begin{aligned}
\text{XPM-MASS}(s) &= \text{XPM-CAPACITY}(O(s \log s)) && \text{(Proposition 2.5)} \\
&\subseteq \text{XPM-TIME}(O(nO(1)^{O(s \log s)})) && \text{(Theorem 4.8)} \\
&= \text{XPM-TIME}(O(ns^{O(s)}))
\end{aligned}$$

□

4.4 Space and Time Hierarchies

We obtain pointer machine space and time hierarchies that are analogous to hierarchies for RAMs and Turing machines. The space hierarchies for pointer machines follow from the space hierarchy for Turing machines (Hartmanis *et al.*, 1965a; Sipser, 1980) and Theorems 4.4 and 4.5.

Corollary 4.10 *If $s_2(n)$ is capacity-constructible, then there is a language $L \subseteq \{0,1\}^*$ such that some pointer machine recognizes L within capacity $O(s_2(n))$, but for any function $s_1(n) = o(s_2(n))$, no pointer machine recognizes L within capacity $O(s_1(n))$.*

Proof.

$$\begin{aligned}
\text{XPM-CAPACITY}(s_1) &\subseteq \text{XTM-SPACE}(s_1) && \text{(Theorem 4.5)} \\
&\subset \text{XTM-SPACE}(s_2) && \text{(Hartmanis } et\ al.\text{, 1965a)} \\
&= \text{XPM-CAPACITY}(O(s_2)) && \text{(Theorem 4.4)}
\end{aligned}$$

□

Corollary 4.11 *If $s_2(n)$ is mass-constructible, then there is a language $L \subseteq \{0,1\}^*$ such that some pointer machine recognizes L within mass $O(s_2(n))$, but for any function $s_1(n) = o(s_2(n))$, no pointer machine recognizes L within mass $O(s_1(n))$.*

Proof.

$$\begin{aligned}
\text{XPM-MASS}(s_1) &= \text{XTM-SPACE}(s_1 \log s_1) && \text{(Theorem 4.5)} \\
&\subset \text{XTM-SPACE}(s_2 \log s_2) && \text{(Hartmanis } \textit{et al.}\text{, 1965a)} \\
&= \text{XPM-MASS}(O(s_2)) && \text{(Theorem 4.4)}
\end{aligned}$$

□

Corollary 4.12 *If $t(n)/\log t(n)$ is mass-constructible, then PM-TIME($t(n)$) is strictly included in PM-MASS($t(n)$).*

Proof.

$$\begin{aligned}
\text{PM-TIME}(t) &\subseteq \text{PM-MASS}(O(t/\log t)) && \text{(Halpern } \textit{et al.}\text{ (1986))} \\
&\subseteq \text{PM-MASS}(t/\log t) && \text{(Theorem 4.1)} \\
&\subset \text{PM-MASS}(t) && \text{(Corollary 4.11).}
\end{aligned}$$

□

To prove the time hierarchy for pointer machines, we need the following useful result:

Lemma 4.13 *For every pointer machine M there is a pointer machine M' with $|\Delta'| = 2$ that simulates M in real time and constant factor overhead in space.*

Proof. Let Δ be the pointer alphabet of M, with $d = |\Delta|$, and let $\Delta' = \{\alpha, \beta\}$ be the pointer alphabet of M'. For each node x in M and each δ_i in Δ, let δ_i point from x to y_i. For each node x in M there is a chain of $d+1$ nodes in M', connected by α pointers, with one distinguished node x' corresponding to x.

Consider one such chain in M'. From each node in the chain, except x', use the β pointer to point to the appropriate y_i' corresponding to y_i (see Figure 4.3). □

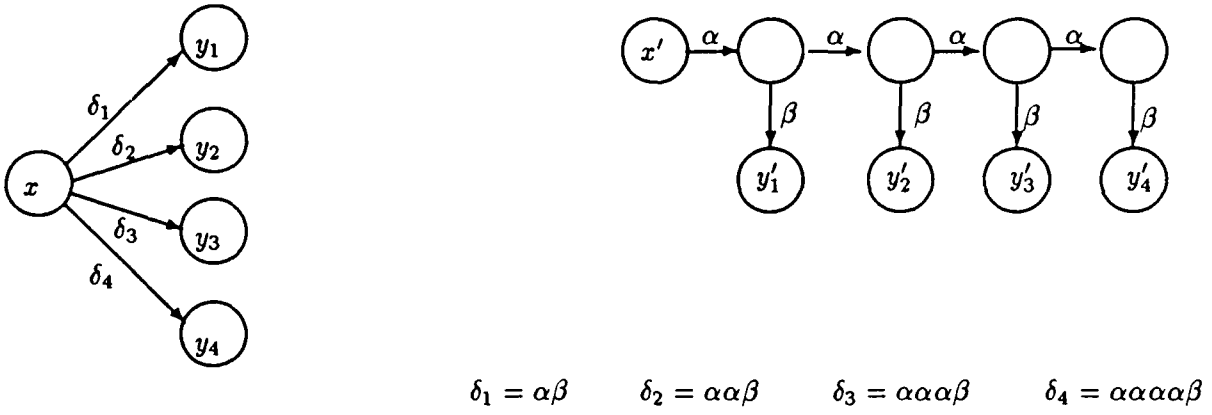


Figure 4.3: Simulating a pointer machine with $|\Delta'| = 2$.

Theorem 4.14 *If $t_2(n)$ is time-constructible by a pointer machine, then there is a language $L \subseteq \{0,1\}^*$ such that some pointer machine recognizes L within time $O(t_2(n))$, but for any function $t_1(n) = o(t_2(n))$, no pointer machine recognizes L within time $O(t_1(n))$.*

Proof. Using techniques Cook and Reckhow (1973) applied to exhibit a time hierarchy for RAMs, we construct a universal pointer machine that can diagonalize over time $t_1(n)$ computations in time $O(t_2(n))$ (see also Hartmanis and Hopcroft, 1971).

It is straightforward to show how a pointer machine program can be encoded with alphabet $\{0,1\}$: we first encode each pointer machine instruction with $\{0,1\}$. To properly encode instruction labels referenced in **if** instructions, we may assume that every instruction is labeled and that the instructions are labeled sequentially with the first instruction labeled 1. We then use a unary encoding for the instruction label referenced in an **if** instruction. To encode the pointer

machine program, we concatenate the encodings of the instructions comprising the program. Let M_w be the pointer machine whose encoding is w.

By Lemma 4.13 we may assume without loss of generality that for the machine to be simulated, $|\Delta| = 2$.

For every w, let w_i be the encoding in w of the instruction of M_w labeled i. Let $k(M_w) = \max_i\{|w_i|\}$. Thus $k(M_w)$ is a constant that depends only on M_w. Define diagonalization language L as follows: if M_w with input w halts in time $t_2(|w|)/k(M_w)$, then w is in L if and only if M_w does not accept w. If M_w does not halt in time $t_2(|w|)/k(M_w)$, then w is not in L.

Now suppose some pointer machine M recognizes L in time $ct_1(n)$, where c is a constant depending on L. Since $t_1 = o(t_2)$, there is a w sufficiently long such that $ct_1(|w|) < t_2(|w|)/k(M_w)$ and M_w accepts exactly the same language as M: add a sufficient number of **accept** instructions immediately after an **accept** instruction in the program of M and call the encoding of this new pointer machine w. By our definition of L, $w \in L$ (i.e., w is accepted by M_w) if and only if w is not accepted by M_w; so we have a contradiction. Thus no pointer machine accepts L in time $O(t_1(n))$.

We now construct a universal pointer machine M that recognizes L in time $O(t_2(n))$. The key issues involved in the construction of such a machine are that it (1) constructs an appropriate simulator of M_w in time $O(|w|)$, (2) simulates M_w in linear time, and (3) keeps track of the elapsed time of the simulated machine so that it can stop after $t_2(n)/k(M_w)$ simulated steps.

M goes through three stages: initialization, preprocessing, and simulation.

1. **Initialization**:

M first creates eight *type nodes* with an appropriate *type pointer* corresponding to each. Each type node corresponds to a distinct pointer machine instruction type.

M creates two *alphabet nodes* (since $|\Delta| = 2$) with an appropriate *alphabet pointer* corresponding to each. M uses the alphabet nodes to designate appropriate arguments for the pointer machine instructions, as described below.

Initialization takes a constant amount of time.

2. **Preprocessing:**

M reads w and decodes it, creating a node for each instruction. Call the set of instruction nodes the *instruction list.* Each instruction node has the following additional pointers:

a. a *successor* pointer to the next instruction node.

b. an *instruction* pointer to the appropriate type node.

c. an *argument* pointer to an *argument structure* described below.

d. a *jump* pointer, for an **if** instruction, to point to the instruction node to which control could be passed.

e. a *beginner* pointer to the initial node (used to set the *jump* pointer appropriately).

For each instruction node, M uses the type pointer to determine the instruction type of the instruction node. M uses alphabet pointers in a similar manner.

The *argument structure* is necessary for the **create**, **center**, **assign**, and **if** instructions. These instructions have arguments that are words in Δ^*. For any instruction with argument $W \in \Delta^*$ such that $|W| = j$, the argument structure contains a chain of j nodes, each node having a pointer to one of the alphabet nodes so that M can access the chain to determine the argument W.

To set the *jump* pointers, M makes a second pass through the instruction nodes and the input word w. For each node representing an **if** instruction, M returns to the initial node with the *beginner* pointer (recording the current instruction node with a pointer). If the instruction specifies a conditional jump to instruction m, M uses the unary encoding of m in w to move the *jump* pointer to the m^{th} instruction node. In this way, setting the *jump* pointers takes time $O(|w|)$.

During preprocessing, M also creates a linked list of $t_2(|w|)$ nodes called the *counter* to keep track of elapsed time of the simulation. This is possible since t_2 is time-constructible. There is one special pointer to the last node in this list. M also creates a linked list of $k(M_w)$ nodes to serve as the *auxiliary counter*; the use of the auxiliary counter is explained below.

Three additional pointers from the initial node keep track of the simulation. The *execute* pointer points from the initial node to the node representing the instruction being simulated. The *counter* pointer points to a node in the counter to indicate how much time has elapsed. The *auxiliary* pointer points to a node in the auxiliary counter.

Preprocessing takes $O(t_2(|w|)) + O(|w|) = O(t_2(|w|))$ time. Figure 4.4 shows the result of initialization and preprocessing (not all pointers or nodes are shown).

3. **Simulation:**

To begin the simulation, M resets the execution and counter pointers to the beginnings of the instruction list and counter, respectively. M also resets the input tape head to the leftmost nonblank tape cell. M maintains a *simulation arena* to keep track of the simulation, so M creates a node to serve as the initial node of M_w in the simulation arena.

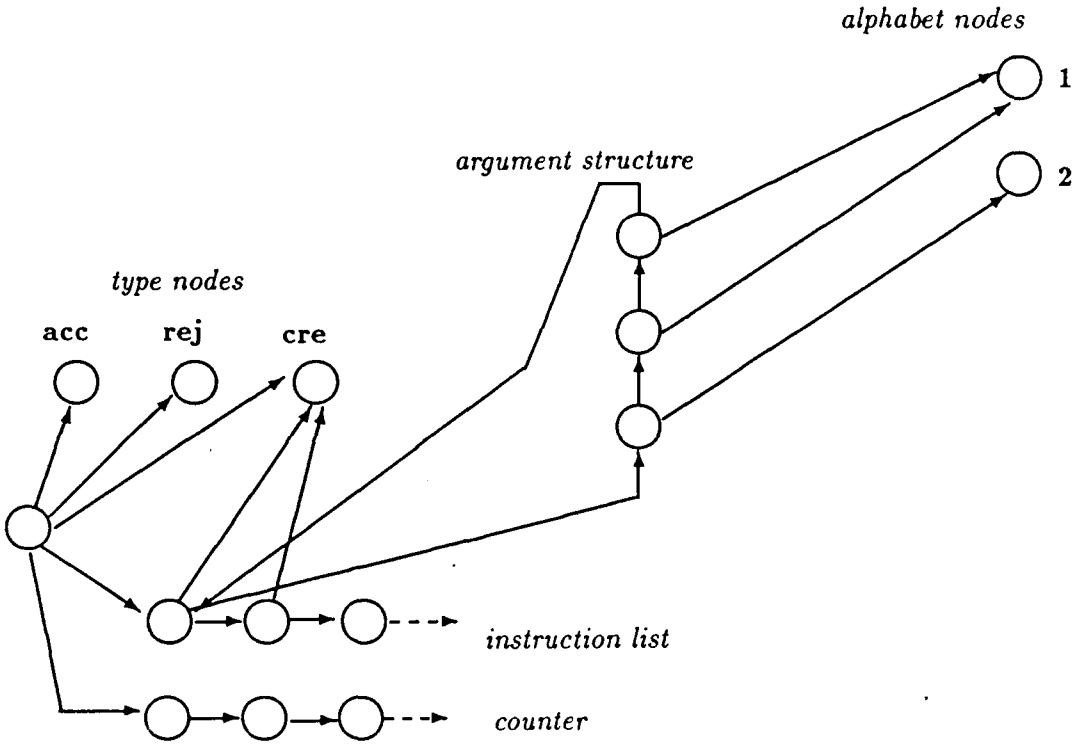


Figure 4.4: Preparation for simulating M_ω

Using the instruction pointer, M accesses the first instruction node, decodes the instruction, and executes the instruction in the simulation arena. M then accesses the next instruction node by following the successor or jump pointer, as appropriate. It continues in this manner, decoding instructions and executing them in the simulation arena, until it reaches an accept or reject instruction node.

The initial node serves as the point of reference for the simulation. It has pointers to the current instruction node, the current counter node, and the current center of M_w in the simulation arena. There is a pointer from every node in the Δ-structure to the initial node so that it may always be referenced.

For each simulated instruction, M moves the counter pointer across $k(M_w)$ nodes of the counter, using the $k(M_w)$ nodes of the auxiliary counter. Since M has a special pointer to the last counter node, M can compare the special pointer with the counter pointer to determine whether $t_2(|w|)/k(M_w)$ steps have elapsed. If so, then M rejects w. Otherwise, M accepts w if and only if M_w rejects w.

Since the length of each encoded instruction of M_w is at most $k(M_w)$, M can simulate each step of M_w in $c'k(M_w)$ steps, for constant c' depending only on M. For each step, M counts off $k(M_w)$ of the $t_2(|w|)$ counter nodes; therefore, M accepts L in time $O(t_2(|w|))$. □

4.5 Nondeterministic Pointer Machines

Using space equivalence for pointer machines and Turing machines, we obtain a result for pointer machines based on Savitch's (1970) result comparing deterministic and nondeterministic Turing machine space:

Theorem 4.15 *For $s(n) \geq \log n$, every nondeterministic pointer machine of capacity complexity $s(n)$ can be simulated by a deterministic pointer machine of capacity complexity $O(s(n)^2)$.*

Proof.

$$\begin{aligned}
\text{NPM-CAPACITY}(s) &\subseteq \text{NTM-SPACE}(s) && \text{(Theorem 4.5)} \\
&\subseteq \text{TM-SPACE}(s^2) && \text{(Savitch, 1970)} \\
&\subseteq \text{PM-CAPACITY}(O(s^2)) && \text{(Theorem 4.4)}
\end{aligned}$$

□

Corollary 4.16 *For $s(n) \geq (\log n)/\log\log n$, every nondeterministic pointer machine of mass complexity $s(n)$ can be simulated by a deterministic pointer machine of mass complexity $O(s(n)^2 \log s(n))$.*

Proof.

$$\begin{aligned}
\text{NPM-MASS}(s) &= \text{NPM-CAPACITY}(O(s \log s)) && \text{(Proposition 2.5)} \\
&\subseteq \text{PM-CAPACITY}(O(s^2(\log s)^2)) && \text{(Savitch, 1970; } s \log s \geq \log n\text{)} \\
&= \text{PM-MASS}(O(s^2 \log s)) && \text{(Proposition 2.5)}
\end{aligned}$$

□

4.6 Alternating Pointer Machines

One result of Chandra *et al.* (1981) is that every alternating Turing machine running in time t can be simulated by a deterministic Turing machine using space t. The following theorem extends that result to pointer machines. It strengthens the result of Halpern *et al.* (1986) that every deterministic pointer machine running in time t can be simulated by a deterministic pointer machine using $O(t/\log t)$ nodes.

Theorem 4.17 *Every alternating pointer machine running in time $t(n)$ can be simulated by a deterministic pointer machine using $t(n)/\log t(n)$ nodes.*

Proof. We construct deterministic pointer machine M' that simulates alternating pointer machine M. Without loss of generality, assume that M alternates strictly between universal and existential states at each step, and that at every step there are exactly two choices. We first describe how the simulation is accomplished with $O(t)$ nodes, and then we show how to reduce this to $O(t/\log t)$. The simulation does not require t or $t/\log t$ to be constructible: M' begins by assuming $t = 1$; when more nodes are necessary, M' begins the simulation with the value of t incremented by 1. In this way, M' uses the minimum number of nodes necessary.

M' begins by creating a chain of t *computation nodes* and two additional nodes designated ℓ and r. Using the computation node chain M' performs a depth-first traversal of the computation tree of M in a manner analogous to the proof of Theorem 3.2 of Chandra *et al.* (1981). Each computation node records a choice made by M on a particular branch of the computation tree with a *choice pointer* to the ℓ or the r node, indicating a left or right branch, respectively. M' initially sets the choice pointer of every computation node to the ℓ node.

For each branch of the computation tree of M, M' uses the recursive simulation procedure SIMULATE of Halpern *et al.* (1986), referring to the computation node chain to determine the proper instruction sequence. SIMULATE uses $O(t/\log t)$ nodes. When SIMULATE terminates for any particular branch of the computation tree, M' backtracks along the computation node chain, resetting pointers appropriately to specify the next branch to be considered.

This simulation uses ct nodes, for some constant c, since t computation nodes are required. The chain of t computation nodes is equivalent to a Turing machine tape of t cells, each of which holds ℓ or r. We reduce the number of nodes to $O(t/\log t)$ by encoding the chain using the tree and blockset in the proof of Theorem 4.2 (see Figure 4.2).

By Theorem 4.1, we can simulate the alternating pointer machine using $t/\log t$ nodes. □

Corollary 4.18 *Every alternating pointer machine running in time $t(n)$ can be simulated by a deterministic Turing machine using space $t(n)$.*

Proof.

$$\begin{aligned} \text{APM-TIME}(t) &\subseteq \text{PM-MASS}(t/\log t) && \text{(Theorem 4.17)} \\ &= \text{TM-SPACE}(t) && \text{(Theorem 4.5)} \end{aligned}$$

□

We can obtain other results based on the theorems of Chandra *et al.* (1981).

Theorem 4.19 *For $s(n) \geq n$, every nondeterministic pointer machine of capacity complexity $s(n)$ can be simulated by an alternating pointer machine in time $O(s(n)^2)$.*

Proof.

$$\begin{aligned} \text{NPM-CAPACITY}(s) &\subseteq \text{NTM-SPACE}(s) && \text{(Theorem 4.5)} \\ &\subseteq \text{ATM-TIME}(O(s^2)) && \text{(Chandra } et\ al.\text{, 1981)} \\ &\subseteq \text{APM-TIME}(O(s^2)) && \text{(Theorem 4.2)} \end{aligned}$$

□

Corollary 4.20 *For $s(n) \geq n/\log n$, every nondeterministic pointer machine of mass complexity $s(n)$ can be simulated by an alternating pointer machine in time $O((s(n)\log s(n))^2)$.*

Proof.

$$\begin{aligned} \text{NPM-MASS}(s) &= \text{NPM-CAPACITY}(O(s\log s)) && \text{(Proposition 2.5)} \\ &\subseteq \text{APM-TIME}(O((s\log s)^2)) && \text{(Theorem 4.19)} \end{aligned}$$

□

Theorem 4.21 *For $s(n) \geq \log n$, every alternating pointer machine of capacity complexity $s(n)$ can be simulated by a deterministic pointer machine in time $O(1)^{s(n)}$.*

Proof.

$$\begin{aligned}
\text{APM-CAPACITY}(s) &\subseteq \text{ATM-SPACE}(s) && \text{(Theorem 4.5)} \\
&\subseteq \text{TM-TIME}(O(1)^s) && \text{(Chandra } et\ al.\text{, 1981)} \\
&\subseteq \text{PM-TIME}(O(1)^s) && \text{(Theorem 4.2)}
\end{aligned}$$

□

Corollary 4.22 *For $s(n) \geq (\log n)/\log\log n$, every alternating pointer machine of mass complexity $s(n)$ can be simulated by a deterministic pointer machine in time $s(n)^{O(s(n))}$.*

Proof.

$$\begin{aligned}
\text{NPM-MASS}(s) &= \text{NPM-CAPACITY}(O(s \log s)) && \text{(Proposition 2.5)} \\
&\subseteq \text{PM-TIME}(O(1)^{O(s \log s)})) && \text{(Theorem 4.21)} \\
&= \text{PM-TIME}(s^{O(s)})
\end{aligned}$$

□

Chapter 5

Optimal Simulation of Tree Machines by Random Access Machines

We present an optimal on-line simulation of a tree machine of time complexity t by a log-cost RAM of time complexity $O((t \log t)/\log\log t)$. This result is a complement to Loui's (1983) simulation of tree machines by multidimensional Turing machines and Reischuk's (1982) simulation of multidimensional Turing machines by tree machines. We begin by exhibiting a real-time simulation of a tree machine by a unit-cost RAM.

5.1 Simulation by Unit-cost RAMs

Theorem 5.1 *Every tree machine can be simulated by a unit-cost RAM in real-time.*

Proof sketch. We design a unit-cost RAM R that simulates tree machine T with worktape W. R has a *contents memory*, a *parent memory*, and several working registers. Let *contents*(x) (respectively, *parent*(x)) be the register with address x in the contents (respectively, parent) memory. *Contents*(x) at address x contains the contents of cell(x) at location x in the worktape of T. If cell(x) is visited by T, then *parent*(x) contains the worktape location of the parent of cell(x). The working registers are used as temporary storage and to keep track of which cell is currently accessed by T.

R simulates one step of T with a constant number of accesses to the two memories and the working registers. For example, if the head moves from cell(x) to a child of cell(x), then R

computes location $2x$ for the left child or $2x+1$ for the right child with one or two additions and stores x in *parent*$(2x)$ or *parent*$(2x+1)$. Thus to simulate t steps of T takes $O(t)$ time on R. □

5.2 Simulation by Log-cost RAMs

5.2.1 Upper Bound

Using the simulation in Theorem 5.1, we can show that every tree machine can be simulated on-line by a log-cost RAM in time $O(t \log t)$; however, we describe below a more efficient simulation by log-cost RAMs.

For simplicity, we consider tree machines with only one tree worktape, but our results generalize to multiple worktapes. Let T be a tree machine of time complexity t with one worktape W. We show that there is a RAM R that simulates T on-line in time $O((t \log t)/\log\log t)$.

Since this is an on-line simulation, we do not know n or $t(n)$ ahead of time. To solve this problem, we use a technique of Galil (1976), adopted by Loui (1983; 1984a) and Katajainen *et al.* (1988). Let t' be the elapsed time of T (as recorded by R) and let t_e be R's current estimate of the total running time of T. R begins the simulation with $t_e = 2$. When t' exceeds t_e, R doubles t_e and restarts the entire simulation. R continues this process of doubling t_e whenever t' exceeds t_e until the simulation is finished. R records the input in a separate memory as described below so that for each value of $t_e > 2$, it is unnecessary to move the input head until $t' > t_e/2$. We show that for each value of t_e, the time of the simulation is $O(t_e(\log t_e)/\log\log t_e)$. It is easy to show that the sum of the simulation times for all values of t_e is $O(t'(\log t')/\log\log t')$.

We first provide a brief description of the simulation. We choose parameters h and u such that $u = 2^{2h+2} - 1$. We specify the values of h and u later. R has several memories. R maintains

in the *main memory* the entire contents of W. The main memory represents W as overlapping subtrees, called *blocks*. R represents the contents of each block W_x in one register r_x of the main memory. When the worktape head is in a particular block W_x, R represents W_x in the *cache memory*. Step-by-step simulation is carried out in the cache, which represents the block W_x in breadth-first order, one cell of W_x per register of the cache.

Because blocks overlap, when the worktape head exits W_x, it is positioned in the middle of some other block W_y. At this time R packs the contents of the cache back into r_x in the main memory and unpacks the contents of r_y into the cache.

The details of the simulation follow.

Let $W[x,s]$ denote the complete subtree of W of height s rooted at cell(x). A *block* is any subtree $W_x = W[x, 2h+1]$ such that the depth of cell(x) is a multiple of $h+1$. Since a block has height $2h+1$, it contains $2^{2h+2} - 1 = u$ cells. Let the *relative location* of a cell within a block be defined in a manner similar to the location of a cell, where the relative location of the root of the block is 1, the relative locations of its children are 2 and 3, and so on.

Call a block W_p the *parent block* of W_x if cell(p) is the ancestor of cell(x) at distance $h+1$ from cell(x). If W_x is the parent block of W_c, then W_c is a *child block* of W_x. Each block has 2^{h+1} child blocks. The topmost block of W, which contains the root of W, is called the *root block*.

Define the *top half* of a block W_x as $W[x,h]$, and define the *bottom half* of W_x as the remaining cells of the block. Note that the top half of the block W_x is part of the bottom half of W_p, its parent block, so that the blocks overlap. Call the portion of W_x shared by W_p (i.e., the subtree $W[x,h]$) the *common subtree* of W_x and W_p.

R precomputes in separate memories two tables, *half* and *translate*. We explain later how R uses these tables. Here we describe their contents and how they are computed. Let *half*(z) (respectively, *translate*(z)) be the register in *half* (respectively, *translate*) at address z.

Half(z) contains $\lfloor z/2 \rfloor$. To compute *half*, for $z = 1, \ldots, u/2$, R stores z in *half*$(2z)$ and *half*$(2z+1)$.

For $z = 2^{2h+1}, \ldots, u$, *translate*(z) contains $(z \bmod 2^{h+1}) + 2^{h+1}$. R never refers to any register in *translate* with address less than 2^{2h+1}. *Translate* is computed as follows:

$i := 2^{h+1}$
for $z = 2^{2h+1}$ **to** u **do**
    translate$(z) := i$
    $i := i + 1$
    if $i = 2^{2h+2}$ **then** $i := 2^{h+1}$
end

We now show how R simulates the tree machine using the cache. Assume the head of T is currently scanning a cell in block W_x. Let *cache*(z) be the register in the cache with address z and let cell(x, z) be the cell in W_x with relative location z. For each $z = 1, \ldots, u$, register *cache*(z) contains the bit in cell(x, z); for example, *cache*(1) contains the contents of cell$(x, 1)$ = cell(x), the root of W_x. Thus R uses u registers of the cache, each register containing one bit.

While the head of T remains in W_x, R keeps track of the head's location with the *cache address register* in the *working memory*, a memory maintained by R for storing information necessary for miscellaneous tasks. If the cache address register contains z, then cell(x, z) is currently being accessed in T.

To simulate a tree machine operation at cell(x, z), R loads the contents (one bit) of *cache*(z) into AC. Once the contents are in AC, R simulates one step of T by storing either 0 or 1 in *cache*(z).

If the head of T moves to a child of cell(x, z), then the new address for the cache address register, as well as the relative location of the new block cell being read, is either $2z$ or $2z + 1$. With one or two additions, R computes this new address and places it in the cache address register. When the head of T moves to the parent of cell(x, z), the address of the corresponding cache register is $\lfloor z/2 \rfloor$. Because R has no division operation, it accesses table *half* to retrieve the new address in cache.

To describe what happens when the worktape head moves out of the current block, we first show how the blocks are stored in main memory. Main memory is divided into *pages* consisting of $2^{h+1} + 3$ registers each. A page corresponds to a visited block of W. Let *page*(x) be the page representing W_x. Define the address of a page to be the address of the first register in the page. The first register in *page*(x) is the *contents register*. For the page representing the root block, the contents register contains the entire contents of that block. For every other block W_y, the contents register contains the contents of the bottom half of W_y. The contents of cells in a block are kept in breadth-first order; i.e., reading the binary string in the contents register from left to right is equivalent to reading the bottom half of the block it represents in breadth-first order. Initially, all cells of a block contain 0, so all contents registers initially contain 0.

Following the contents register is the *rank register*, containing a number ℓ between 1 and 2^{h+1} indicating that W_x is the ℓ^{th} child of its parent block. The next register is the *parent register*, containing the address of the page representing the parent block of W_x. The next 2^{h+1} registers are the *child registers* of W_x. The m^{th} child register of *page*(x) contains the address of the page representing the m^{th} child block of W_x or 0 if that child block has not been visited (see Figure 5.1).

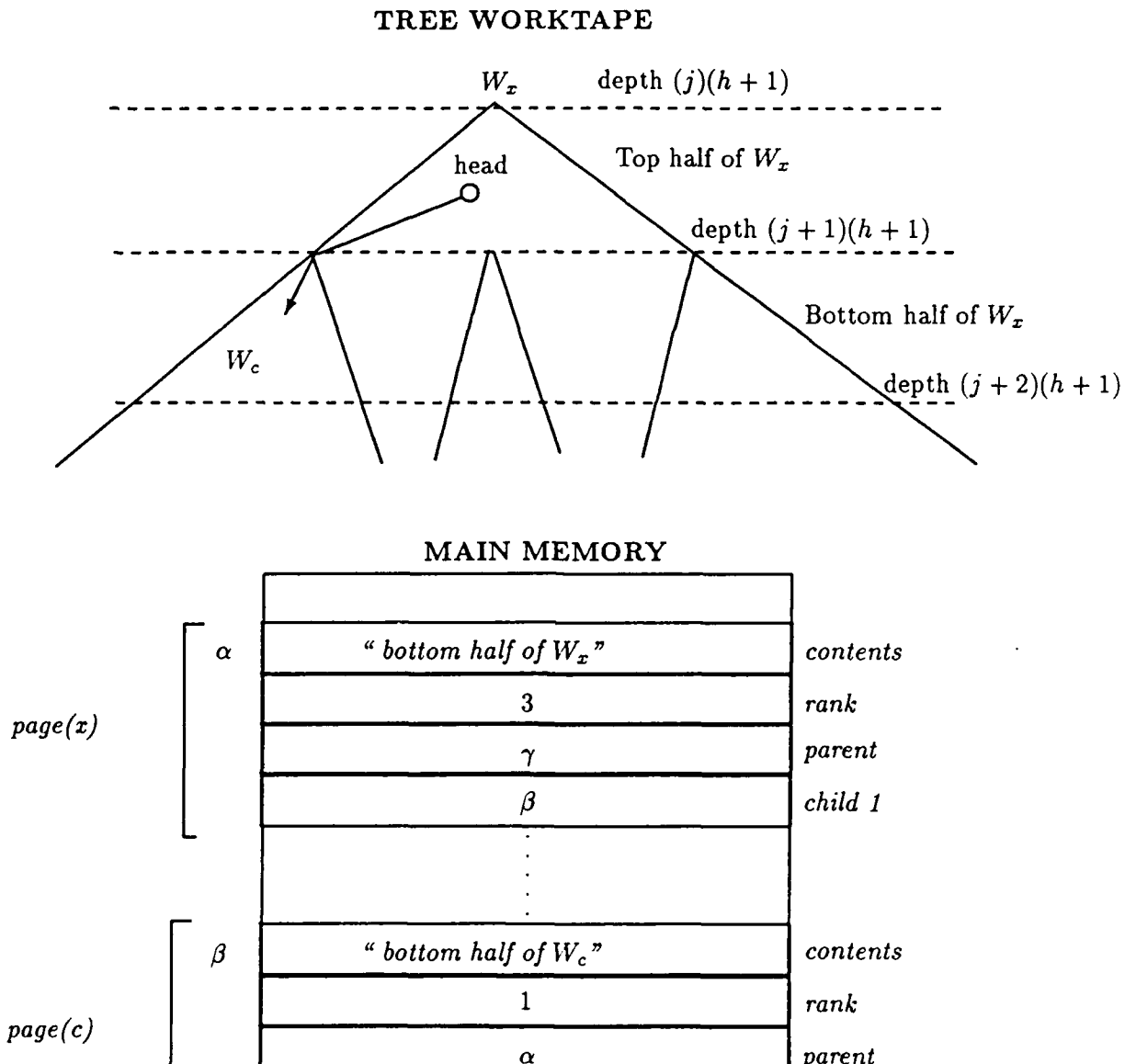


Figure 5.1: Worktape W (head moves from W_x to W_c)

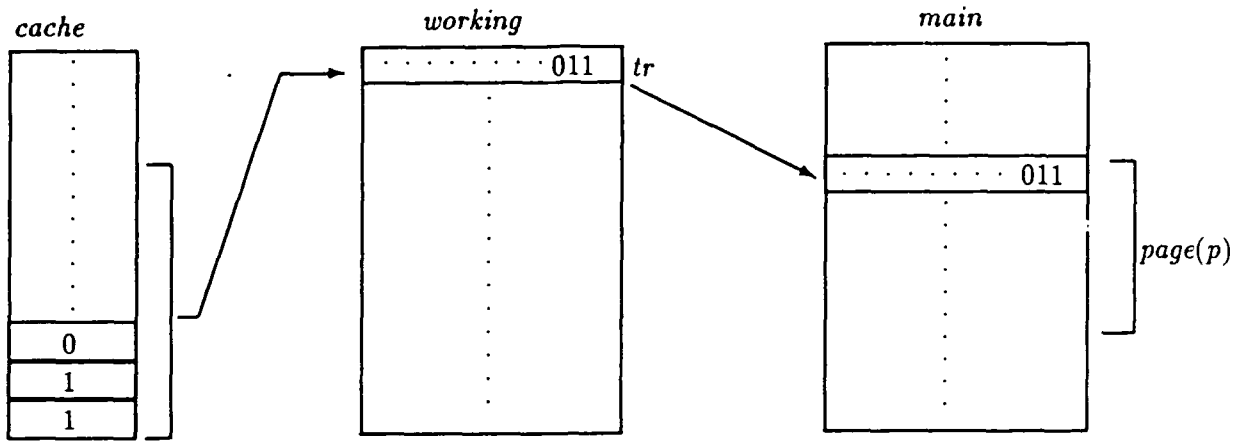


Figure 5.2: Updating *page*(p) in main memory

The first page in main memory corresponds to the root block. Blocks are then stored in the order in which they are visited. The *page address register*, a register in working memory, contains the address of the page in main memory corresponding to the currently accessed block.

Let W_x be the currently accessed block and let W_p be the parent block of W_x. When the tree worktape head moves out of W_x so that it is positioned in the middle of a child block W_c, R makes the proper changes to main memory and loads the cache from the contents register of *page*(c).

In main memory, R updates the contents registers of *page*(x) and *page*(p). To update *page*(x), R packs the contents of the registers of the cache which correspond to the bottom half of W_x into a single register in working memory (call it the *transfer register*, denoted by *tr*). R then copies *tr* into the contents register of *page*(x) via AC (see Figure 5.2).

Updating *page*(p) consists of changing the bits of its contents register corresponding to the common subtree of W_x and W_p. R first saves the contents of the cache that encode the common subtree of W_x and W_c in a portion of working memory, since this information is needed in the cache as the top half of W_c. R also saves the contents of the cache that encode the common

subtree of W_x and W_p. R then loads the contents register of *page(p)* into *tr* and unpacks the contents into the cache. The bits in working memory corresponding to the common subtree of W_x and W_p are then written into their proper locations in the portion of the cache representing the bottom half of W_p. R then packs the contents of the cache into *tr* and copies *tr* into the contents register of *page*(p).

R then determines whether W_c has been visited before by checking the contents of the child register of *page*(x) corresponding to W_c. If the child register contains a valid (i.e., nonzero) address, then R uses that address to access *page*(c). R then unpacks the contents register of *page*(c) into the cache. This action is similar to the manipulation of *page*(p) discussed above. R loads the contents of the common subtree of W_x and W_c saved in working memory into the registers of the cache representing the top half of the block.

If the child register of *page*(x) contains 0, then R allocates a new page to maintain the information on W_c.

R modifies the page address register to reflect the fact that the worktape head is now scanning block W_c. The address currently in this register is that of *page*(x). R writes the address of *page*(c) in main memory to the page address register. R determines from the cache address register the quantity ℓ such that W_c is the ℓ^{th} child of W_x. Then by accessing the ℓ^{th} child register of *page*(x) in the main memory, R can determine the address of *page*(c).

To modify the cache address register to reflect the relative location of the head within block W_c, R first translates the relative location of the leaf cell(x, z) in W_x into its relative location in W_c. Since leaf cell(x, z) in W_x is the same as cell$(c, (z \bmod 2^{h+1}) + 2^{h+1})$ in W_c, R uses the table *translate* described above. Using one or two additions, R then calculates the relative

location in W_c of this cell's left or right child, depending on which branch the worktape head used to exit W_x. R then writes this new relative location into the cache address register.

A similar sequence of operations occurs if the worktape head moves out of a block (and farther) into its parent block instead of into a child block. Then R uses the parent register to determine the address of the page representing the parent block, and R uses the rank register to determine the relative location of the worktape head within the parent block.

As described earlier, R maintains an estimate t_e of the total running time of T. R doubles t_e whenever the elapsed time exceeds t_e and restarts the simulation with this new value. The portion of the input string read by T up to time $t_e/2$ is maintained in R's *input memory* in registers of length h. Input symbols read from time 1 to time h are contained in the first register of input memory, those read from time $h+1$ to time $2h$ are contained in the second register, etc. Each register is unpacked into the *input cache* at its appropriate time, and the input symbols are read by R. After $t_e/2$ steps of the tree machine have been simulated, input is read from the input tape. This new input is stored in the same manner as previous input.

When it is necessary to restart the simulation with a new value of t, R reorganizes the input memory using packs and unpacks so that register lengths reflect the updated value of h.

To simulate tree machines with more than one worktape, R maintains a main memory, a cache, and a working memory for each worktape.

By evaluating the cost of the simulation on a log-cost RAM, we derive the following result.

Theorem 5.2 *Every tree machine running in time $t(n)$ can be simulated on-line by a log-cost RAM running in time $O((t(n)\log t(n))/\log\log t(n))$.*

Proof. Because the blocks have height $2h+1$ and overlap by height $h+1$, whenever the worktape head moves out of a block, it is exactly in the middle of another block; i.e., T will take

at least $h' = h + 1$ steps before its worktape head exits this new block. Since the tree machine computation has at most t steps, the work of updating main memory from cache (packing), loading a new block into the cache (unpacking), and directly simulating h' steps is performed at most t/h' times.

Updating main memory and loading a new block in cache involve the pack and unpack operations and a constant number of accesses to main memory. Registers in main memory have addresses no larger than $(t/h')(2^{h+1}+3)$. Thus accesses to main memory take time $O(\log t + h)$.

By Lemma 2.3, the time for the pack and unpack operations is $O(u \log u)$. By Lemma 2.4, the time to create the tables necessary for these operations is $O(u2^u)$. The time to compute tables *half* and *translate* is $O(u \log u)$.

Simulating one step of the tree machine consists of a constant number of accesses to cache, taking time $O(\log u)$. Thus simulating h' steps takes time $O(h' \log u)$.

Simulating h input operations (those up to step $t/2$) takes time $O(h \log h)$. Recording h input operations (those past step $t/2$) also takes time $O(h \log h)$. Packing and unpacking takes time $O(h \log h)$. Thus the time to simulate $t/2$ input operations and record $t/2$ additional input operations is $(t/h)O(h \log h)$. Reconfiguring the input memory for a new value of t also takes time $(t/h)O(h \log h)$. Building the necessary tables for input simulation and recording takes time $O(h2^h)$.

The total time required for R, then, is

$$(t/h')(O(\log t + h) + O(u \log u) + O(h' \log u)) + O(u2^u) + t \log h.$$

Since $h = O(\log u)$, the total time is

$$O(((t \log t)/\log u) + tu + t \log u + u2^u).$$

Choose h so that $u = (\log t)/\log\log t$. Then the total time for the simulation is $O((t \log t)/\log\log t)$.

□

5.2.2 Lower Bound

We now show that the time bound of Theorem 5.2 is optimal within a constant factor. We begin with an overview of Kolmogorov complexity, which we use to prove the lower bound.

Let σ and τ be strings in $\{0,1\}^*$, and let U be a universal Turing machine. Define the *Kolmogorov complexity* of σ given τ with respect to U, denoted $K(\sigma|\tau)$, as follows: let $\#$ be a symbol not in $\{0,1\}^*$; then $K(\sigma|\tau)$ is the length of β where β is the shortest binary string such that $U(\beta\#\tau) = \sigma$. Informally, $K(\sigma|\tau)$ is the length of the shortest binary description of σ, given τ. If τ is the empty string, then we write $K(\sigma)$ for $K(\sigma|\tau)$.

We say a string σ is *incompressible* if $K(\sigma) \geq |\sigma|$. Note that for all n there are 2^n binary strings of length n, but there are only $2^n - 1$ strings of length less than n. Thus for all n, there is at least one incompressible string of length n.

A useful concept in Kolmogorov complexity is the *self-delimiting string*. For natural number n, let $bin(n)$ be the binary representation of n without leading 0's. For binary string w, let $\overline{w}$ be the string resulting from placing a 0 between each pair of adjacent bits in w and adding a 1 to the end. Thus $\overline{110} = 101001$. We call the string $\overline{bin(|w|)}w$ the *self-delimiting version* of w. The self-delimiting version of w has length $|w| + 2\log(|w| + 1)$. When we concatenate several binary string segments of differing lengths, we can use self-delimiting versions of the strings so that we can determine where one string ends and the next string begins with little additional cost in the length of the concatenated string. Note that in such a concatenation it is not necessary to use a self-delimiting version of the last string segment.

Kolmogorov complexity has recently gained popularity as a method for proving lower bounds. Li and Vitanyi (1988) provide a thorough summary of lower bound (and other complexity-related) results obtained using Kolmogorov complexity.

Theorem 5.3 *There is a tree machine T running in time n such that for any log-cost RAM R, R requires time $t(n) = \Omega((n \log n)/\log\log n)$ to simulate T on-line.*

Proof. Tree machine T has one tree worktape and operates in real time. T's input alphabet is a set of *commands* of the form $\langle e, \psi \rangle$, where $e \in \{0, 1, ?\}$ and ψ indicates whether the worktape head moves to a child or parent of the current cell or remains at the current cell. Suppose T is in a configuration in which the cell x at which the worktape head is located contains e'. On input $\langle e, \psi \rangle$, machine T writes e' onto its output tape, and the worktape head writes e onto cell x if $e \in \{0, 1\}$, but it writes e' (the current contents of x) onto x if $e =?$. At the end of the step the worktape head moves according to ψ. For every n that is a sufficiently large power of 2, we construct a sequence of n tree commands for which R requires time $\Omega((n \log n)/\log\log n)$. As in (Loui, 1983), the string of tree commands is divided into a *filling part* of length $n/2$ and a *query part* of length $n/2$.

Let W be the worktape of T, and let x_0 be the root of W. Let $d = \log(n/8)$. Denote the complete subtree of W of height d whose root is x_0 by W_d. Let $N = n/8$. We consider the complexity of the simulation in terms of N.

We fill W_d with an incompressible string τ of length $2N - 1$ such that τ can be retrieved by a depth-first traversal of W_d. This takes time $4N - 4$ on T. We move the worktape head four more times (without writing) so that the total length of the filling part is $n/2$.

The query part consists of a series of *questions*. A *question* is a string of $2d = 2\log N$ tree commands that causes the worktape head to move from the root x_0 of the tree worktape to a

cell at depth d and back to x_0 without changing the contents of the worktape. As the head visits each cell during a question, T outputs the contents of that cell. T processes $2N/\log N$ questions $Q_1, Q_2, \ldots$ during the query part. Thus the query part takes time $4N = n/2$. We show that after each question Q_j, there is a question Q_{j+1} such that R takes time $\Omega((\log^2 N)/\log\log N)$ to process Q_{j+1}, and Theorem 5.3 follows.

Assume that R has just processed question Q_j. Let $P(N)$ be the maximum time necessary to process any possible next question. We show that some next question takes time $\Omega((\log^2 N)/\log P)$. Consequently, by definition, $P = \Omega((\log^2 N)/\log P)$. To determine a lower bound on P, we consider two cases:

(1) $P \leq \log^2 N$; hence, $\log P \leq 2\log\log N$. Thus we have the following:

$$\begin{aligned} P &\geq c(\log^2 N)/\log P, \text{ for some constant } c \quad (\text{since } P = \Omega((\log^2 N)/\log P) \\ &\geq c(\log^2 N)/(2\log\log N); \end{aligned}$$

(2) $P \geq \log^2 N$.

In either case, $P = \Omega((\log^2 N)/\log\log N)$.

We first determine $\hat{t}$, the sum over all possible next questions q, of the time required for R to process q.

Divide worktape W into $S = (\log N)/(2\log P)$ *sections*, each of height $2\log P$. For $s = 0, 1, \ldots, S-1$, there are P^{2s+2} exit points (*bottom cells*) in section s. We refer to any initial segment of a question as a *partial question* and the portion of the question that is processed while the worktape head is in one section as a *subquestion* (see Figure 5.3). To compute $\hat{t}$, we compute for $s = 0, 1, \ldots, S-1$ the total time $\hat{t}_s$ required for R to process all possible subquestions in section s. Since the depth of W_d is $\log N$, there are N possible next questions. Each of the P^{2s+2} bottom cells of section s is visited during N/P^{2s+2} of these questions.

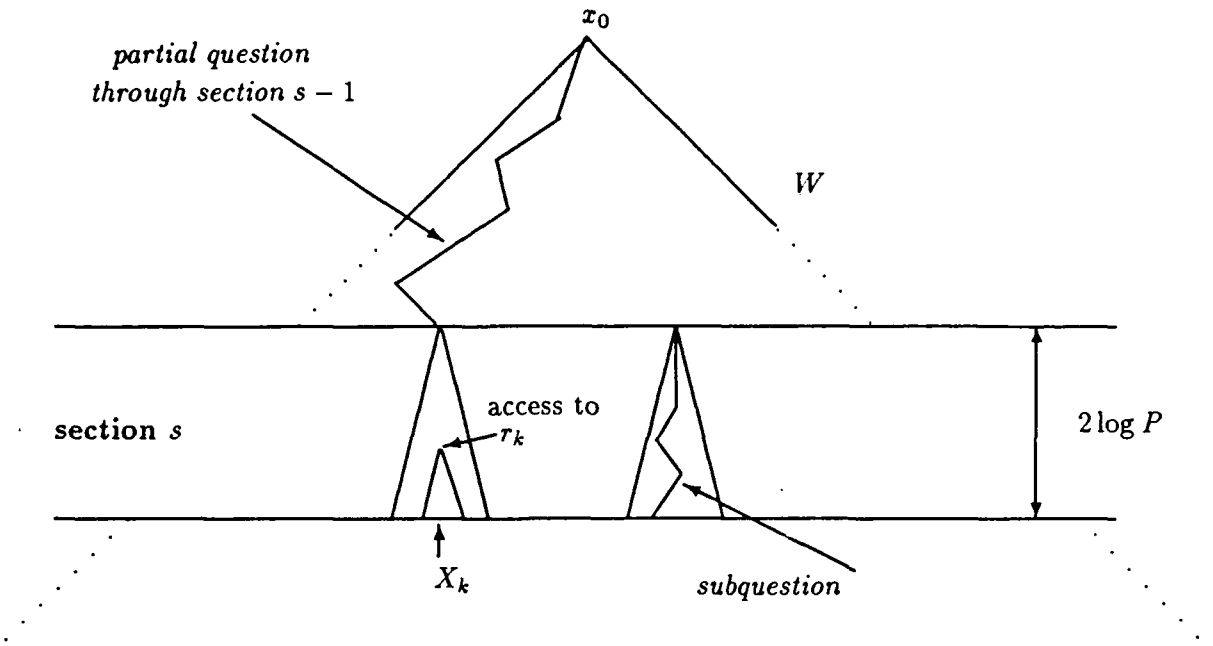


Figure 5.3: Processing section s of worktape W

Let σ_s be the string defined by the contents of the bottom cells of section s, from left to right; clearly, $|\sigma_s| = P^{2s+2}$.

Lemma 5.4 *The string σ_s is incompressible up to a term of $O(s \log P)$; i.e., $K(\sigma_s) \geq |\sigma_s| - O(s \log P)$.*

Proof. The incompressible string τ, which gives the contents of W, can be specified by a string composed of the following segments:

1. a self-delimiting string encoding this discussion ($O(1)$ bits)

2. a self-delimiting version of a binary string of length $K(\sigma_s)$ that specifies σ_s ($K(\sigma_s) + O(s \log P)$ bits)

3. self-delimiting versions of the values of s and P ($O(\log s) + O(\log P)$ bits)

4. a string specifying the bits in τ but not in σ_s ($2N - 1 - P^{2s+2}$ bits).

Thus $K(\tau) \leq K(\sigma_s) + (2N - 1 - P^{2s+2}) + O(s \log P)$. But $K(\tau) \geq 2N - 1$; therefore, $K(\sigma_s) \geq P^{2s+2} - O(s \log P)$. □ **Lemma 5.4**

Lemma 5.5 *If* $\ell \geq 1$, *then* $\sum_{i=1}^{\ell} \log i \geq (1/2)\ell \log \ell$.

Proof. For all i such that $1 \leq i \leq \ell$, clearly $(i - 1)(\ell - i) \geq 0$; hence $i(\ell - i + 1) \geq \ell$. Consequently,

$$\begin{aligned} \sum_{i=1}^{\ell} \log i &= (1/2) \sum_{i=1}^{\ell} (\log i + \log(\ell - i + 1)) \\ &= (1/2) \sum_{i=1}^{\ell} \log(i(\ell - i + 1)) \\ &\geq (1/2) \sum_{i=1}^{\ell} \log \ell \\ &= (1/2)\ell \log \ell. \end{aligned}$$

□ **Lemma 5.5**

Lemma 5.6 *For* $s = 1, 2, \ldots, S - 1$, *the maximum number of distinct registers accessed during the processing of all partial questions through section* $s - 1$ *is at most* $4P^{2s+1}/\log P$.

Proof. Let $C = 4P/\log P$. By Lemma 5.5, for P sufficiently large, $\sum_{i=1}^{C} \log i \geq P$. The processing of each partial question through section $s - 1$ could involve no more than C distinct registers; otherwise, because of the total cost of addresses of registers, R would exceed time P for some next question. There are P^{2s} different partial questions possible through section $s - 1$, so there are no more than $4P^{2s+1}/\log P$ distinct registers accessed for all possible partial questions. □ **Lemma 5.6**

Let us consider a particular section s. Let $r_1, r_2, \ldots, r_m$ be the registers, in order of increasing address, that R accesses to produce the same output that T produces when its worktape head is in section s, excluding those registers accessed to process partial questions through section $s-1$. The address of r_i is at least i. To compute a lower bound on $\hat{t}_s$, we assess for each i the contribution to $\hat{t}_s$ of accessing r_i.

To determine the contribution of r_i to $\hat{t}_s$, we calculate the minimum number of possible questions for which R accesses r_i. For every bottom cell v, let q_v be the subquestion that causes T to visit cell v of the tree worktape. For $1 \leq i \leq m$, let X_i be the set of bottom cells x of section s such that $x \in X_i$ if R accesses r_i to process q_x (see Figure 5.3). Thus if T visits a cell in X_i when processing a question in section s, R accesses register r_i when processing the same question. We say that r_i *operates* on the bottom cells in X_i. Since T visits one cell of X_i while processing one of N/P^{2s+2} possible questions, R accesses r_i during the processing of at least $|X_i|(N/P^{2s+2})$ possible questions.

For $1 \leq i \leq m$, the total access time for register r_i in section s is at least the product of $\log i$ (since the address of r_i is at least i), $|X_i|$ (the number of bottom cells that r_i operates on), and N/P^{2s+2} (the number of questions during which one of these bottom cells is visited). Summing the time incurred by access to each register yields:

$$\hat{t}_s \geq \sum_{i=1}^{m} (\log i)|X_i|(N/P^{2s+2}). \tag{5.1}$$

Using Lemma 5.8 below, we can determine a lower bound for $\hat{t}_s$, but we first introduce the following technical lemma.

Lemma 5.7 (Loui, 1984b [Section 4]) *Let J and M be integers such that $M \geq J$. A sorted J-member subset of $\{0, \ldots, M\}$ can be represented with no more than $2J\log(M/J) + 4J + 2$ bits.*

Let $h = (1/7)P^{2s+1}$.

Lemma 5.8 $\sum_{i=h}^{m} |X_i| \geq (1/23)P^{2s+2}$.

Proof. Assume that the conclusion is false. Then $r_1, \ldots, r_{h-1}$ operate on at least $(22/23)P^{2s+2}$ bottom cells in section s. We can specify the string σ_s as follows: we obtain the bits of $X_h, \ldots, X_m$ explicitly. We obtain the other bits of σ_s by simulating R on each partial question to a bottom cell of section s not in $\bigcup_{k=h}^{m} X_k$. On each such partial question, R uses only registers $r_1, \ldots, r_{h-1}$ and registers accessed in sections $1, \ldots, s-1$. Thus σ_s can be specified with a string composed of the following segments:

1. a self-delimiting string encoding the program of R and this discussion ($O(1)$ bits)

2. self-delimiting versions of the addresses and initial contents of registers accessed in sections $1, \ldots, s-1$ (at most $8P^{2s+2}/\log P + O(s \log P)$ bits – by Lemma 5.6, at most $4P^{2s+1}/\log P$ registers are required, and for each register, the contents and the address could each require P bits.)

3. self-delimiting versions of the addresses and initial contents of $r_1, \ldots, r_{h-1}$ ($(2/7)P^{2s+2} + O(s \log P)$ bits)

4. a string specifying positions of cells in X_k for $k \geq h$ (we use Lemma 5.7 with $J = (1/23)P^{2s+2}$ and $M = P^{2s+2}$; this requires at most $(14/23)P^{2s+2}$ bits. The encoding used to achieve Lemma 5.7 is such that the beginning and end of this string can easily be determined.)

5. a string specifying the contents of cells in X_k for $k \geq h$ (at most $(1/23)P^{2s+2}$ bits).

This means that the number of bits needed to specify σ_s is at most $(151/161)P^{2s+2} + O(P^{2s+2}/\log P) < P^{2s+2} - O(s \log P)$ for sufficiently large P. Thus we have a contradiction of Lemma 5.4. □ **Lemma 5.8**

Thus we have:

$$\begin{aligned}
\hat{t}_s &\geq \sum_{i=1}^{m}((\log i)|X_i|(N/P^{2s+2}) && \text{(Inequality 5.1)} \\
&\geq \sum_{i=h}^{m}((\log i)|X_i|(N/P^{2s+2})) \\
&\geq (N/P^{2s+2})(\log h)\sum_{i=h}^{m}|X_i| \\
&\geq (N/P^{2s+2})(\log h)(1/23)P^{2s+2} && \text{(Lemma 5.8)} \\
&\geq (1/23)N((2s+1)\log P - \log 7) && \text{(definition of } h\text{)} \\
&\geq (1/23)Ns \log P.
\end{aligned}$$

Now sum $\hat{t}_s$ over all s to compute a lower bound for $\hat{t}$, the total time required for R to process all possible next questions:

$$\begin{aligned}
\hat{t} &= \sum_{s=0}^{S-1}\hat{t}_s \\
&\geq \sum_{s=0}^{S-1}((1/23)Ns \log P) \\
&\geq (1/23)N(\log P)((\log^2 N)/(4\log^2 P) - O((\log N)/\log P)) \\
&\geq (1/92)((N \log^2 N)/\log P - O(N \log N)).
\end{aligned}$$

Since there are N questions, we divide $\hat{t}$ by N to derive the average time needed by R to process the next question, $\Omega((\log^2 N)/\log P)$. Some next question must require time greater than or equal to this average time. Since P is the maximum time for some next question, $P \geq \Omega((\log^2 N)/\log P)$; hence, $P = \Omega((\log^2 N)/\log\log N)$.

Thus for each question Q_j, we can choose a next question Q_{j+1} that takes time $\Omega((\log^2 N)/\log\log N)$. Since the query part has $N/(2\log N)$ questions, our choice of questions means that the query part takes time $t = (N/(2\log N))\Omega((\log^2 N)/\log\log N)) = \Omega((N\log N)/\log\log N)$. The entire simulation takes at least time t. Since $N = n/8$, the lower bound holds for n as well. □ **Theorem 5.3**

Because the lower bound proof considers only the time involved in accessing registers, the lower bound holds for RAMs with more powerful instructions, such as boolean operations or multiplication.

5.3 Implications for Log-cost RAMs and Unit-cost SRAMs

The lower bound of Theorem 5.3 implies a lower bound on simulating a unit-cost SRAM by a log-cost RAM. We present the theorem in terms of pointer machines instead of unit-cost RAMs.

Theorem 5.9 *There is a pointer machine P running in time $O(n)$ such that for any log-cost RAM R that simulates P on-line, R requires time $\Omega((n\log n)/\log\log n)$.*

Proof. Let T be the tree machine described in Theorem 5.3. Let P be a pointer machine that simulates T. It is straightforward to show that every tree machine can be simulated by a pointer machine in real time. T runs in time n, so P runs in time $O(n)$. Now assume there is a log-cost RAM that simulates P on-line in time $o((n\log n)/\log\log n)$. We thus have an on-line simulation of a tree machine of time complexity n by a log-cost RAM running in time $o((n\log n)/\log\log n)$. But we know from Theorem 5.3 that the lower bound on such a simulation is $\Omega((n\log n)/\log\log n)$; hence we have a contradiction. □

Chapter 6

Relationships between Multidimensional Turing Machines and RAMs

6.1 Simulation of Multidimensional Turing Machines by RAMs

6.1.1 Simulation by Log-cost RAMs

By composing our simulation in Subsection 5.2.1 of a tree machine by a log-cost RAM with Reischuk's (1982) simulation of a d-dimensional Turing machine by a tree machine, we obtain an on-line simulation of a d-dimensional Turing machine of time complexity t by a log-cost RAM running in time $O((5^{d \log^* t} t \log t)/\log\log t)$. But we improve this upper bound with a direct simulation.

Theorem 6.1 *Every d-dimensional Turing machine running in time $t(n)$ can be simulated on-line by a log-cost RAM running in time $O(t(n)(\log t(n))^{1-(1/d)}(\log\log t(n))^{1/d})$.*

Proof. We design a log-cost RAM R that simulates d-dimensional Turing machine M. Since this is an on-line simulation, we use the procedure of the simulation in Subsection 5.2.1, doubling t as necessary.

For simplicity, assume M has one worktape; our results generalize to d-dimensional Turing machines with more than one worktape. Let $s = ((\log t)/\log\log t)^{1/d}$. Partition the worktape of M into boxes with side length s. Let $base(i)$ be the base cell in box i. For every cell x in a

box, there are 3^d boxes that contain cells with coordinates that all differ from the coordinates of x by at most s; i.e., there are 3^d boxes that are within distance s of cell x.

For box i, if $base(i) = (i_1, i_2, \ldots, i_d)$, let $index(i) = i_d t^{d-1} + i_{d-1} t^{d-2} + \ldots + i_1$. R stores the contents of box i (s^d bits) in the register in *main memory* with address $index(i)$. Step-by-step simulation is carried out in the *cache*. R conducts the simulation in t/s phases, each of s steps of M. For each phase: R unpacks the contents of the 3^d boxes that are currently within distance s of the worktape head (the head remains within these boxes during the phase); R simulates M for s steps; and R packs the contents of the cache back to main memory. Using precomputed values of $t, t^2, \ldots, t^{d-1}$ (R can compute each of these values in time $O(t)$), R quickly computes $index(i')$ from $index(i)$ when box i' is adjacent to box i.

For each phase, R takes time $O(\log t)$ to access main memory, $O(\log t)$ to compute the address of registers in main memory representing the new blocks needed in the cache, $O(s \log s)$ to simulate s steps in the cache, $O(s^d \log s)$ to pack and unpack the appropriate registers (Lemma 2.3), and $O(s2^s) = o(t)$ to build the appropriate tables (Lemma 2.4). Thus the total time for the simulation is:

$$(t/s)(O(\log t) + O(s \log s) + O(s^d \log s)) + O(s2^s)$$

$$= O(((t \log t)/s) + ts^{d-1} \log s)$$

$$= O(t(\log t)^{1-(1/d)}(\log \log t)^{1/d}).$$

□

6.1.2 Simulation by Unit-cost RAMs and SRAMs

Schönhage (1980) proved that every multidimensional Turing machine can be simulated by a unit-cost SRAM in real-time. Because a unit-cost RAM can simulate a unit-cost SRAM in

real-time, it is clear that a unit-cost RAM can simulate a d-dimensional Turing machine in real-time; however, we can do better by adapting a result of Grigor'ev:

Theorem 6.2 *For $t(n) \geq n(\log n)^{1/d}$, every d-dimensional Turing machine running in time t can be simulated on-line by a unit-cost RAM in time $O(t(n)/(\log t(n))^{1/d})$.*

Proof sketch. Grigor'ev (1979) presented an off-line version of this result. He adapted the simulation of a one-dimensional Turing machine by a unit-cost RAM (Hopcroft *et al.*, 1975). We briefly sketch his simulation. Let M be a d-dimensional Turing machine running in time t, and let R be the RAM simulator. Call a box of M of side length $c(\log t)^{1/d}$, where constant $c < 1$, a *block*. A d-dimensional Turing machine is *block-respecting* if its worktapes are divided into blocks and its heads pass block boundaries only at times that are integer multiples of $c(\log t)^{1/d}$. Grigor'ev showed that M could be converted into a block-respecting machine M' running in time $O(t)$. We construct R to simulate M'.

R computes the $O(t^c)$ possible configurations for M'. For each of these configurations, R simulates M' for $c(\log t)^{1/d}$ steps to determine its next configuration. Computing the configurations and next configurations take time $O(t^c(\log t)^{1/d})$. The actual simulation of M' consists of $O(1)$ table lookups for every $c(\log t)^{1/d}$ steps of M'. Block adjacency information is maintained using the pyramidal structure employed by Schönhage (1980) in his real-time simulation of a multidimensional Turing machine by a pointer machine. This structure allows R to determine which blocks of M' have already been visited by worktape heads. It can be maintained by R in real-time (that is, $O(1)$ steps for each block visit).

Grigor'ev's simulation can be converted into an on-line simulation using the same procedures that Galil (1976) used to convert the simulation of Hopcroft *et al.* (1975) to an on-line simulation. We do not know the value of n or $t(n)$ ahead of time, so we use the procedure of

Subsection 5.2.1, doubling t as necessary. Another problem is that M' is block-respecting, but it may need to read input symbols at time steps that are not integer multiples of $c(\log t)^{1/d}$. Galil solved this problem with a *super block-respecting* machine, that is, a machine that is block-respecting and only reads inputs at a time which is a multiple of $c(\log t)^{1/d}$. Galil showed that M' could be modified so that it was super block-respecting and still run in time $O(t)$. This modification involved introducing appropriate delays between inputs. The same technique works in this case. Hence we have R simulate the super block-respecting version of M', and we now have an on-line simulation. □

6.2 Simulation of RAMs by Multidimensional Turing Machines

6.2.1 Simulation of Log-cost RAMs

Loui (1983) provided an upper bound on simulating log-cost RAMs by multidimensional Turing machines: he showed that every log-cost RAM of time complexity t can be simulated on-line by a d-dimensional Turing machine in time $O(t^{1+(1/d)}/\log t)$.

We can prove that there is a log-cost RAM R running in time t such that every d-dimensional Turing machine requires time $\Omega((t^{1+(1/d)}(\log\log t)^{1+(1/d)})/(\log t)^{2+(1/d)})$ to simulate R on-line. Suppose, to the contrary, that every log-cost RAM can be simulated on-line in time $o((t^{1+(1/d)}(\log\log t)^{1+(1/d)})/(\log t)^{2+(1/d)})$. Combining this simulation with the optimal on-line simulation of tree machines by log-cost RAMs outlined in Subsection 5.2.1, we obtain a simulation of tree machines by d-dimensional Turing machines. Applying this simulation to the real-time tree machine T described in Loui's (1983) proof of a lower bound on on-line simulation of tree machines by multidimensional Turing machines, we obtain a d-dimensional

Turing machine that simulates T on-line in time $o(n^{1+(1/d)}/\log n)$, which contradicts the lower bound established by Loui.

Here we derive a stronger lower bound for simulating log-cost RAMs by multidimensional Turing machines. We first introduce Lemma 6.3. This result relies on the fact that there is a fixed constant c such that for all binary strings σ and τ,

$$K(\sigma) \leq 2|\sigma| + c \quad \text{and} \quad K(\sigma) \leq K(\sigma|\tau) + K(\tau) + c,$$

where $K()$ denotes Kolmogorov complexity, defined in Subsection 5.2.2. The constant c is used in the lemma.

Lemma 6.3 (Loui, 1983) *Let $g \geq 1$ and let σ be an incompressible string of length $n > 8(c+g)$. For every set of g strings $\{\tau_1, \tau_2, \ldots, \tau_g\}$ each of length at most $n/(4g)$, $K(\sigma|\tau_1\#\tau_2\#\cdots\tau_g) > n/4$.*

Theorem 6.4 *There is a log-cost RAM R running in time $O(n)$, where n is the input length, such that for any d-dimensional Turing machine M that simulates R on-line, M requires time $\Omega(n^{1+(1/d)}/(\log n(\log\log n)^{1+(1/d)}))$.*

Proof. As in Loui (1983), we construct a hard input string consisting of a filling part and a query part. The *filling part* comprises an incompressible binary string x of length $n/(2\log\log n)$. Delimiters are added to x so that R can easily read x in pieces of length $(\log n)/2$. This string is followed by some "dummy bits" to pad out the filling part so that its total length is $n/2$. R processes the filling part in *cycles*. In each cycle, R reads the next $(\log n)/2$ bits into a cache and packs these bits into one register of main memory; so R eventually writes x into the first $n/(\log n \log\log n)$ registers of main memory.

R computes the tables necessary for packing after it reads the first piece of x. By maintaining a counter during the reading of the first piece, R can determine the value of $(\log n)/2$ and use this value to build the tables.

R takes time $O(\log n \log\log n)$ to determine the value of $(\log n)/2$. By Lemma 2.4 with $u = (\log n)/2$, R takes $o(n)$ time to precompute the tables necessary for packing. In each cycle, R takes $O(\log n \log\log n)$ time for packing (Lemma 2.3) and $O(\log n)$ time for access to one of the first $n/(\log n \log\log n)$ registers of main memory. Each of $n/(\log n \log\log n)$ cycles takes time $O(\log n \log\log n)$, so the filling part takes time $O(n)$.

A *question* for R is a string of the form $a\$b$, where a is a binary string of length $\log n$ that specifies an address in main memory, and b is a binary string of length $\log\log n$ that specifies the position of a particular bit within that register. To process question $a\$b$, R accesses register $r(a)$ in main memory and outputs the bit at position b in $\langle a\rangle$, the contents of $r(a)$, without changing the contents of main memory. After R reads a into the accumulator, R uses an indirect read to obtain $\langle a\rangle$ and unpacks $\langle a\rangle$ into the cache. R then reads b into the accumulator, accesses the register in the cache corresponding to position b, and outputs that bit. The time to read a and b into the accumulator and to unpack the contents of $r(a)$ is $O(\log n \log\log n)$, so R takes time $O(\log n \log\log n)$ on one question.

The *query part* is a sequence of $n/(\log n \log\log n)$ questions $Q_1, Q_2, \ldots$, so R takes time $O(n)$ to process the query part. Now we show how to choose questions so that M spends time $\Omega((n/\log\log n)^{1/d})$ to process each Q_j.

Let M have h access heads on one worktape. For $j \geq 1$, consider the configuration of M immediately before reading the first symbol of Q_j. Let $B_1, \ldots, B_h$ be the boxes of side length $(n/(4c'h\log\log n))^{1/d}$ centered at the heads in this configuration, where constant c'

depends on M and is chosen later. These boxes hold all cells accessed by M during the next $(n/(4c'h\log\log n))^{1/d}/2$ steps. Let y_i be a binary encoding of the contents of B_i and z_i be a binary encoding of the relative position of access head i in B_i. Let $|B_i|$ be the volume of B_i, and let $|y_i|$ and $|z_i|$ be the respective lengths of y_i and z_i. For c' sufficiently large, $|y_i| \leq c'|B_i|$ and $|z_i| \leq c'|B_i|$ for every i. If M could process every possible question Q_j with the heads remaining in $B_1 \cup \cdots \cup B_h$, then from the string $y_1\#\cdots\#y_h\#z_1\#\cdots\#z_h$, only a small constant amount of additional information (a binary description of this discussion) would be necessary to generate x. Thus $K(x|y_1\#\cdots\#y_h\#z_1\#\cdots\#z_h) = O(1)$, but by Lemma 6.3, $K(x|y_1\#\cdots\#y_h\#z_1\#\cdots\#z_h) > n/(8\log\log n)$, and we have a contradiction.

Therefore, since for each j there exists a Q_j such that some head spends time $\Omega(n/(4h\log\log n))^{1/d}/2$ to exit $B_1 \cup \cdots \cup B_h$ when M processes question Q_j, the time spent by M on the query part is at least $(n/(\log n\log\log n))(n/(4h\log\log n))^{1/d}/2 = \Omega(n^{1+(1/d)}/(\log n(\log\log n)^{1+(1/d)}))$. □

6.2.2 Simulation of Unit-cost RAMs

Wagner and Wechsung (1986) show that every unit-cost RAM of time complexity t can be simulated by a d-dimensional Turing machine in time $O(t^{2+(1/d)})$. We improve their result with the following theorem.

Theorem 6.5 *For $d \geq 2$, every unit-cost RAM of time complexity $t(n)$ can be simulated on-line by a d-dimensional Turing machine in time $O(t(n)^2\log t(n))$.*

Proof. We design a d-dimensional Turing machine M that simulates unit-cost RAM R. Since we do not know n or $t(n)$ ahead of time, we use the procedure of Subsection 5.2.1, doubling t as necessary.

Each register accessed by R is represented on the d-dimensional tape by a box of cells called a *record*. We refer to the coordinates of the base cell of a record as the coordinates of the record. Conceptually, records correspond to nodes in a height-balanced binary tree. A depth-first (preorder) traversal of the tree would produce a list of the records sorted by the addresses of the registers that the records represent.

Each record x contains a register address, register contents, coordinates of the parent of x (in binary), coordinates of the left and right children of x (in binary), and balancing information. On a separate worktape, M keeps the coordinates of the next unused record available for insertion into the tree.

In t steps, R accesses at most t registers, so M creates at most t records. Thus M keeps the contents of all registers used in the computation of R in a large box called the *storage area*. The storage area consists of at most t records. Let the volume of the storage area be V.

Since a unit-cost RAM can double the contents of a register at each step, after t steps, the maximum value in any register (and the maximum register address) is 2^t, which can be represented by t bits. The coordinates of any record can be represented using $O(\log V)$ bits, and the balancing information requires only a constant number of bits. Thus each record is a box of volume $O(t + \log V)$. Since there are at most t records, $V = O(t^2 + t \log V)$. We want to make the volume of the storage area as small as possible, so choose $V = O(t^2)$. The side length of the storage area is $O(t^{2/d})$.

To move to record x whose coordinates are specified in record y, M writes the coordinates of record x on a separate worktape. M then uses the procedure of Lemma 2.1, so the time to move a worktape head to record x is $O(t^{2/d})$.

Each step of R consists of $O(1)$ accesses to memory . To simulate an access to register $r(a)$, M performs a search through the tree of records. Since the tree is height-balanced, M visits $O(\log t)$ records. For each record x visited, M accesses $O(t)$ cells to compare a with the register address in x. The time to move the head to the next record is $O(t^{2/d})$. The time to access the contents portion of a record is $O(t)$. The time to perform an arithmetic operation on operands of length $O(t)$ is $O(t)$. Since $d \geq 2$, the total time taken for each accessed record is $O(t)$. To keep the tree height-balanced requires adjustments to $O(\log t)$ records. Thus the time for each step of R is $O(t \log t)$. The total time for the simulation is $O(t^2 \log t)$. □

Theorem 6.6 *There is a unit-cost RAM R running in time n such that for any d-dimensional multihead Turing machine M, M requires time $\Omega(n^{1+(1/d)}/\log n)$ to simulate R on-line.*

Proof. Let T be the tree machine described in the lower bound proof of Loui (1983). Let R be a unit-cost RAM that simulates T in real time (Theorem 5.1). Since T runs in time n, R runs in time $O(n)$. Now assume there is a d-dimensional Turing machine that simulates R on-line in time $o(n^{1+(1/d)}/\log n)$. We thus have an on-line simulation of a tree machine of time complexity n by a d-dimensional Turing machine running in time $o(n^{1+(1/d)}/\log n)$. But we know from Loui (1983) that the lower bound on such a simulation is $\Omega(n^{1+(1/d)}/\log n)$; hence we have a contradiction. □

6.2.3 Simulation of Unit-cost SRAMs

Wagner and Wechsung (1986) describe a simulation of a unit-cost SRAM of time complexity t by a d-dimensional Turing machine in time $O(t^{1+(1/d)} \log t)$. We present the following improvement:

Theorem 6.7 *For $d \geq 2$, every unit-cost successor RAM of time complexity $t(n)$ can be simulated on-line by a d-dimensional Turing machine in time $O(t(n)^{1+(1/d)}(\log t(n))^{1/d})$.*

Proof. We design a d-dimensional Turing machines M that simulates pointer machine P. The theorem follows from real-time equivalence of pointer machines and unit-cost SRAMs. Since we do not know n or $t(n)$ ahead of time, we again employ the technique of repeatedly doubling t and starting over as necessary. By Lemma 4.13, we can assume that P has a pointer alphabet of size 2.

As in the simulation of Theorem 6.5, M maintains information about each node of P in a *record*, a box of volume $O(\log t)$. Let record i represent the i^{th} node created by P. The record whose base cell is the origin contains the coordinates of the record representing the center node and the coordinates of the next available record (to represent the next node to be created by P). For every other record in M, record i contains the coordinates of the records representing the two nodes pointed to by node i in P. Since P creates at most t nodes, M maintains information about all nodes in a large box of volume $O(t \log t)$ and side length $O((t \log t)^{1/d})$. Call this large box the *storage area.*

Each step of R consists of $O(1)$ accesses to nodes in the Δ structure. M moves a worktape head to a record in the storage area as described in Lemma 2.1, taking time proportional to the side length of the storage area. Thus each access can be simulated by M in time $O((t \log t)^{1/d})$. So the entire simulation takes time $O(t(t \log t)^{1/d})$. □

Theorem 6.8 *There is a unit-cost successor RAM R running in time $O(n)$ such that for any multihead d-dimensional Turing machine M that simulates R on-line, M requires time $\Omega(n^{1+(1/d)}/\log n)$.*

Proof sketch. It is straightforward to show that every tree machine can be simulated by a pointer machine, and thus by a unit-cost SRAM, in real time; therefore we can apply the proof technique of Theorem 6.6. We assume the contrary and present an on-line simulation of a tree

machine of time complexity n by a d-dimensional Turing machine in time $o(n^{1+(1/d)}/\log n)$, which contradicts the lower bound result of Loui (1983). □

Chapter 7

Open Problems

7.1 Improving Pack and Unpack Routines for Log-cost RAMs

We have used the *pack* and *unpack* routines of Katajainen *et al.* (1988) for several results. These routines take time $O(b2^b + b\log b)$ time on a RAM. A good question is whether these routines can be improved; that is, whether a log-cost RAM can compute the b-bit representation of an integer $n < 2^b$, or the numerical value of a b-bit string, in $O(b\log b)$ time. Clearly, if the b-bit representation is to be in b registers, then $\Omega(b\log b)$ time is necessary to access and read each of b different registers; but is it possible to improve the $O(b2^b)$ time needed to create the tables used in the routines?

We can extend this packing problem as follows: How fast can a log-cost RAM pack b registers each containing w bits into a single register? Call this the $b - w$ *packing problem.* It is evident that $\Omega(b\log b + bw)$ time is necessary, since the RAM needs $\Omega(b\log b)$ time to write down the addresses of b distinct registers and $\Omega(bw)$ time to access b registers each containing w bits. We can adapt the algorithms of Katajainen *et al.* (1988) to show that the $b - w$ packing problem can be solved in time $O(bw2^{bw} + b\log b + bw)$; the only major change is to ensure that the proper values are contained in the first two registers of the *origin* table. The $O(bw2^{bw})$ term is needed to construct the *origin*, *lshift*, and *rshift* tables.

If the $b - w$ packing problem could be solved in time $O(b\log b + bw)$, then storing arbitrary n-bit inputs would take $O(n\log^* n)$ time on a log-cost RAM. This upper bound would match

the lower bound on storing n-bit inputs presented by Schönhage (1988). Let *pack*(b,w,x) pack into AC the contents of the b registers starting at $r(x)$ with each register containing w bits. The following algorithm, *nstore*, stores an n-bit input in optimal time, if the $b - w$ packing problem is solvable in time $O(b \log b + bw)$:

```
procedure nstore(n, a)
* stores an n-bit number from input into a single register (n + a - 1) in memory

if n = 1 then
    read input bit
    store bit in register(a - 1)
else
    for i := 1 to n/log n do
        nstore(log n, i)
    endfor
    pack(n/log n,log n,log n)
    store contents of accumulator in register (n + a - 1)
endif
end nstore
```

Let $t(n)$ be the time necessary for a log-cost RAM R to execute *nstore* on an n-bit number. For an n-bit number, R takes time $(n/\log n)t(\log n)$ to make the recursive calls to *nstore*. If the $b - w$ packing problem can be solved in time $O(b \log b + bw)$, then the call to *pack* takes time $O((n/\log n)\log n)$. Access to memory takes time $O(\log n)$. Thus analysis of this algorithm gives us (for some constants $k_1, \ldots, k_4$):

$$\begin{aligned} t(1) &= k_1 \\ t(n) &= (n/\log n)t(\log n) + k_2((n/\log n)\log n) + k_3 \log n \\ &= (n/\log n)t(\log n) + k_4 n, \end{aligned}$$

so $t(n) = O(n \log^* n)$.

Another effect of efficient $b - w$ packing would be a strengthening of the lower bound on the on-line simulation of a log-cost RAM by a d-dimensional Turing machine. By Theorem 6.4, we have a lower bound of $\Omega(n^{1+(1/d)}/(\log n(\log\log n)^{1+(1/d)}))$. Theorem 6.4 used an incompressible

string of length $n/(2\log\log n)$, and the query part comprised only $O(n/(\log n \log\log n))$ questions. With efficient $b-w$ packing, we could use an incompressible string of length $n/(2\log^* n)$, and the query part could consist of $O(n/(\log n \log^* n))$ questions. The resulting lower bound would be $\Omega(n^{1+(1/d)}/(\log n(\log^* n)^{1+(1/d)}))$.

7.2 Time versus Space and Determinism versus Alternation

One goal in our research has been to reproduce results for pointer machines already known for Turing machines. We presented time and space hierarchy theorems and several complexity results for nondeterministic and alternating pointer machines.

There are still several open questions in this area. As we have mentioned, we would like to duplicate the time vs. mass result of Halpern *et al.* (1986) for the capacity space measure for pointer machines; i.e., we would like to to show that every pointer machine running in time t can be simulated by a pointer machine running in capacity $O(t/\log t)$. Such a result would be important for at least two reasons. It would provide further evidence that capacity is the proper measure of space complexity for pointer machines, and it would further establish the machine-independence of the time vs. space result for Turing machines (Hopcroft *et al.*, 1977). We believe that this result is possible,but it appears that the approach taken by Halpern *et al.* will not provide the desired result.

One approach is to answer another open problem about pointer machines. Dymond and Tompa (1985) showed that every deterministic Turing machine running in time t can be simulated by an alternating Turing machine running in time $t/\log t$. We would like to show that adding the property of alternation to pointer machines yields a similar result; namely, that every deterministic pointer machine running in time t can be simulated by an alternating pointer

machine running in time $O(t/\log t)$. We believe that this is possible, although proving it may be difficult.

Suppose PM-TIME(t) $\subseteq$ APM-TIME($O(t/\log t)$). Then because APM-TIME(t) $\subseteq$ PM-CAPACITY($O(t)$) (Theorem 4.17), it would follow that PM-TIME(t) $\subseteq$ PM-CAPACITY($O(t/\log t)$). Thus this alternating vs. deterministic pointer machine time result would imply the desired pointer machine time vs. capacity result.

The obvious approach to showing that alternating pointer machines are faster than deterministic pointer machines is to adapt the proof of Dymond and Tompa. The difficulty is that the key to their proof is a pebble game on the computation graph of a Turing machine. A similar computation graph for a pointer machine does not seem to have the necessary properties to exploit the same pebble game. It may be that the pointer machine computation graph must be constructed in a more clever manner, or perhaps some other properties of the more straightforward computation graph can be used to prove this result. Another approach is to avoid explicit construction of the computation graph altogether as in Halpern *et al.* (1986).

7.3 Lower Bound on Simulation of Multidimensional Turing Machines

We are attempting to find a tight lower bound for the on-line simulation of multidimensional Turing machines by log-cost RAMs. We believe that a lower bound of $\Omega(t(\log t)^{1-(1/d)})$ is possible. Although this lower bound would not match the current upper bound of $O(t(\log t)^{1-(1/d)}(\log\log t)^{1/d})$ (Theorem 6.1), it would provide more insight into the possibilities and limitations of dynamic representation of arrays within a set of registers.

Our current approach to this problem is to use Kolmogorov complexity, as in Theorem 5.3, where we established a lower bound on simulating a tree machine by a log-cost RAM. We

describe a d-dimensional machine M running in real time that we believe cannot be simulated by any log-cost RAM in time $o(t(\log\ t)^{1-(1/d)})$.

Again we have a filling part and a query part. For the filling part, M fills a region A of its worktape with an incompressible string x of length $O(n)$ such that the distance from the origin to any cell on the boundary of A is $O(n^{1/d})$. A question drives the worktape head from the origin to a boundary cell of A, and back to the origin. The only other movement restriction on the head during a question is that when the head is moving toward the boundary of A, it must always move away from the origin. Thus a question takes time $O(n^{1/d})$ on M. The query part consists of $O(n^{1-(1/d)})$ questions. Our goal is to show that for every RAM R that simulates M, after i questions, there is always an $(i+1)^{st}$ question on which R takes time $\Omega(n^{1/d}(\log n)^{1-(1/d)})$. The argument below leads us to believe that such a question exists.

We construct a question q as follows: RAM R begins processing q by accessing register r_1. We choose the initial portion of q so that R is forced to access some different register r_2 as soon as possible. We then choose the next portion of q so that R must access another register as soon as possible. We continue in this manner until we have constructed the entire question.

Say R accesses, in order, registers $r_1, r_2, \ldots, r_m$ to process q. Let a_i be the portion of the incompressible string x output by R after R accesses r_i but before R accesses r_{i+1}. For $i = 1, \ldots, m$, let ℓ_i be the length of a_i. Note that $\sum_{i=1}^{m} \ell_i = n^{1/d}$. Since we chose q so that it forced an access to a new register as soon as possible, a_i must be the shortest portion of x that R could output on this access to r_i. Thus the length of the contents of r_i at the time it is accessed (call this length L_i) is $\Omega(\ell_i^d)$ (see Figure 7.1).

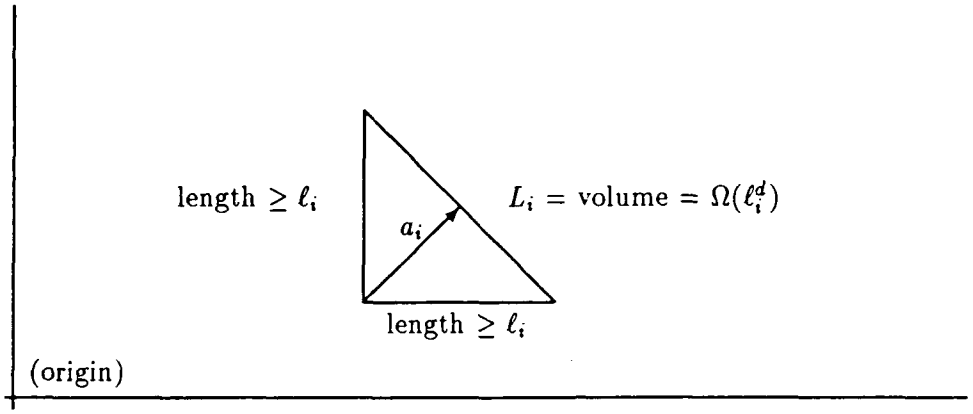


Figure 7.1: Region of worktape that r_i could process

Let m^* be the number of distinct registers among $r_1, r_2, \ldots, r_m$. The time t_q for R to answer the question q is $\Omega(m^* \log m^*)$ to specify m^* distinct register addresses plus $\Omega(\sum_{i=1}^{m}(\ell_i)^d)$. By Jensen's inequality, $\sum_{i=1}^{m}(\ell_i)^d \geq m((\sum_{i=1}^{m}(\ell_i))/m)^d = m(n^{1/d}/m)^d$.

We would like to be able to show that among all the questions we could construct in the above manner, there is at least one question where $m = m^*$; i.e., where all registers accessed are distinct. It might be possible to show that if such a question does not exist, then some group of registers specify more than their share of information about x, hence the string x is compressible. If we could prove the existence of such a question, then we could claim:

$$t_q \geq m \log m + m(n^{1/d}/m)^d = m \log m + nm^{1-d}.$$

The right side is minimized when $m = (n/\log n)^{1/d}$. Thus $t_q = \Omega((n^{1/d}(\log n)^{1-(1/d)})$. This would give us our lower bound.

Unfortunately, we have not yet been able to apply an incompressibility argument to prove the existence of a question where $m = m^*$, nor do we know whether such an argument is possible. We may need to focus on some other aspect of the construction of the questions. In

any case, we believe this approach or one similar will yield the desired lower bound, and we shall continue to investigate the problem.

References

L. M. Adleman and M. C. Loui (1981), Space-bounded simulation of multitape Turing machines. *Math. Systems Theory*, 14:215–222.

A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir (1987), A model for hierarchical memory. In *Proc. 19th Ann. ACM Symp. on Theory of Computing*, pages 305–314.

A. V. Aho, J. E. Hopcroft, and J. D. Ullman (1974), *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company.

V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzev (1970), On economical construction of the transitive closure of a directed graph. *Soviet Math Dokl.*, 11(5):1209–1210.

A. Borodin, M. J. Fischer, D. G. Kirkpatrick, N. A. Lynch, and M. Tompa (1981), A time-space tradeoff for sorting on non-oblivious machines. *J. Comput. System Sci.*, 22:351–364.

A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer (1981), Alternation. *J. Assoc. Comput. Mach.*, 28:114–133.

S. A. Cook and R. A. Reckhow (1973), Time bounded random access machines. *J. Comput. System Sci.*, 7:354–375.

P. W. Dymond (1977), *Complexity Relationships among some Models of Computation.* Master's thesis, University of Toronto.

P. W. Dymond and S. A. Cook (1980), Hardware complexity and parallel computation. In *Proc. 21st Ann. Symp on Foundations of Computer Science*, pages 360–372.

P. W. Dymond and M. Tompa (1985), Speedups of deterministic machines by synchronous parallel machines. *J. Comput. System Sci.*, 30:149–161.

M. J. Fischer (1975), Lecture notes (# 20,21) for Course 6.853, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

M. Fürer (1984), Data structures for distributed counting. *J. Comput. System Sci.*, 28:231 – 243.

Z. Galil (1976), Two fast simulations which imply some fast string matching and palindrome-recognition algorithms. *Information Processing Letters*, 4(4):85–87.

D. Y. Grigor'ev (1977), Imbedding theorems for Turing machines of different dimensions and Kolmogorov algorithms. *Soviet Math. Dokl.*, 18:588–592.

D. Y. Grigor'ev (1979), Time complexity of multidimensional Turing machines. *Zapiski naučnych seminarov*, 88:47–55.

J. Y. Halpern, M. C. Loui, A. R. Meyer, and D. Weise (1986), On time versus space III. *Math. Systems Theory*, 19:13–28.

J. Hartmanis (1971), Computational complexity of random access stored program machines. *Math. Sys. Theory*, 5(3):232–245.

J. Hartmanis and J. E. Hopcroft (1971), An overview of the theory of computational complexity. *J. ACM*, 18(3):444–475.

J. Hartmanis, P. M. Lewis II, and R. E. Stearns (1965a), Hierarchies of memory limited computations. In *IEEE Conf. Record on Switching Circuit Theory and Logical Design*, pages 179–190.

J. Hartmanis and R. E. Stearns (1965b), On the computational complexity of algorithms. *Trans. Amer. Math. Soc.*, 117:285–306.

F. C. Hennie (1966), On-line Turing machine computations. *IEEE Transactions on Electronic Computers*, EC-15(1):35–44.

F. C. Hennie and R. E. Stearns (1966), Two-tape simulation of multitape Turing machines. *J. Assoc. Comput. Mach.*, 13(4):533–546.

J. Hopcroft, W. Paul, and L. Valiant (1975), On time versus space and related problems. In *Proc. 16th Ann. Symp. on Foundations of Computer Science*, pages 57–64.

J. Hopcroft, W. Paul, and L. Valiant (1977), On time versus space. *J. Assoc. Comput. Mach.*, 24:332–337.

J. E. Hopcroft and J. D. Ullman (1979), *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company.

J. Katajainen, J. V. Leeuwen, and M. Penttonen (1988), Fast simulation of Turing machines by random access machines. *SIAM J. Comput.*, 17:77–88.

A. N. Kolmogorov (1965), Three approaches to th quantitative definition of information. *Problems in Information Transmission*, 1:1–7.

M. Li and P. M. B. Vitanyi (1988), Two decades of applied Kolmogorov complexity. To appear in *Handbook of Theoretical Computer Science* (J. van Leeuwen, Managing Ed-

itor), North-Holland. Preliminary version in *Proc. 3rd IEEE Structure in Complexity Theory Conf.*, pages 80–101, 1988.

M. C. Loui (1982), Simulations among multidimensional Turing machines. *Theoret. Comput. Sci.*, 21:145–161.

M. C. Loui (1983), Optimal dynamic embedding of trees into arrays. *SIAM J. Comput.*, 12:463–472.

M. C. Loui (1984a), Minimizing access pointers into trees and arrays. *J. Comput. System Sci.*, 28(3):359–378.

M. C. Loui (1984b), The complexity of sorting on distributed systems. *Inform. and Control*, 60:70–85.

D. R. Luginbuhl and M. C. Loui (1988), *Hierarchies and Space Measures for Pointer Machines.* Technical Report UILU-ENG-88-2245, University of Illinois at Urbana-Champaign. Submitted for publication to *Information and Computation.*

M. C. Loui and D. R. Luginbuhl (1989), *Efficient On-Line Simulations of Tree Machines and Multidimensional Turing Machines by Random Access Machines.* Technical Report UILU-ENG-89-2222, University of Illinois at Urbana-Champaign. Submitted for publication to *Journal of the ACM.*

W. J. Paul (1979), On time hierarchies. *J. Comput. System Sci.*, 19:197–202.

W. J. Paul, N. Pippenger, E. Szemeredi, and W. Trotter (1983), On determinism versus non-determinism and related problems. In *Proc. 24th Ann. Symp. on Foundations of Computer Science*, pages 429–438.

W. J. Paul, E. J. Prauß, and R. Reischuk (1980a), On alternation. *Acta Inform.*, 14:243–255.

W. J. Paul and R. Reischuk (1980b), On alternation II. *Acta Inform.*, 14:391–403.

W. Paul and R. Reischuk (1981), On time versus space II. *J. Comput. System Sci.*, 22:312–327.

W. J. Paul, J. I. Seiferas, and J. Simon (1981), An information-theoretic approach to time bounds for on-line computation. *J. Comput. System Sci.*, 23:108–126.

N. Pippenger (1982), Probabilistic simulations. In *Proc. 14th Ann. ACM Symp. on Theory of Computing*, pages 17–26.

N. Pippenger and M. J. Fischer (1979), Relations among complexity measures. *J. Assoc. Comput. Mach.*, 19:361–381.

K. R. Reischuk (1982), A fast implementation of a multidimensional storage into a tree storage. *Theoret. Comput. Sci.*, 19:253–266.

W. Savitch (1970), Relationships between nondeterministic and deterministic tape complexities. *J. Comput. System Sci.*, 4:177–192.

A. Schönhage (1980), Storage modification machines. *SIAM J. Comput.*, 9(3):490–508.

A. Schönhage (1988), A nonlinear lower bound for random-access machines under logarithmic cost. *J. Assoc. Comput. Mach.*, 35(3):748–754.

M. Sipser (1980), Halting space-bounded computations. *Theoret. Comput. Sci.*, 10:335–338.

R. J. Solomonoff (1964), A formal theory of inductive inference, Part 1 and Part 2. *Inform. and Control*, 7:1–22,224–254.

C. Slot and P. van Emde Boas (1988), The problem of space invariance for sequential machines. *Inform. and Comput.*, 77:93–122.

P. van Emde Boas (1989), Space measures for storage modification machines. *Information Processing Letters*, 30(2):103–110.

K. Wagner and G. Wechsung (1986), *Computational Complexity.* D. Reidel Publishing Company.

J. Wiedermann (1983), Deterministic and nondeterministic simulation of the RAM by the Turing machine. In *Information Processing 83 (IFIP)*, pages 163–168.

C. K. Yap (1987), *An Introduction to the Theory of Complexity Classes, Volume 1.* To be published by Oxford University Press.

Vita

David R. Luginbuhl w[redacted] He graduated in 1976 from Satellite High School in Satellite Beach, Florida. He attended Florida State University, where he majored in mathematics with emphasis in computer science. While at Florida State, he was initiated into Phi Eta Sigma, Phi Beta Kappa, and Phi Kappa Phi. He graduated summa cum laude in June, 1979. In June, 1980, he was commissioned a Second Lieutenant in the United States Air Force.

From October, 1980 to October, 1983, Lieutenant Luginbuhl was assigned to the 20th Missile Warning Squadron and 1020th Computer Services Squadron at Eglin AFB, Florida as a radar software analyst. He was responsible for software controlling radar operation and missile warning functions of the AN/FPS-85 phased array radar system.

In December of 1981, Lieutenant Luginbuhl received the M.S. degree in mathematics with emphasis in computer science from Florida State University.

From October, 1983 to August, 1986, Lieutenant Luginbuhl was assigned to Headquarters, Tactical Air Command at Langley AFB, Virginia. There he developed computer programming applications and software to support analysis of Tactical Air Forces issues. While at Langley AFB, he was promoted to his present rank of Captain. He left that assignment for his current assignment: to pursue a Ph.D. in Computer Science at the University of Illinois.

Upon graduation, Captain Luginbuhl will be an instructor at the Air Force Institute of Technology at Wright-Patterson AFB, Ohio.

Captain Luginbuhl is an Air Force ROTC Distinguished Graduate. He has received two Air Force Commendation Medals and the Air Force Organizational Excellence Award.

Captain Luginbuhl is an active member of the Urbana Free Methodist Church. His extracurricular activities include participation in Graduate InterVarsity Christian Fellowship, running, and playing volleyball. He also enjoys terrorizing his dog, Ascii.